Configuring and Extending Agents

Last modified: 2026-09-13 00:11:34 (PDT)

Coding agents adapt to a project through configuration files, instruction prompts, tool definitions, and plugins. This chapter explains how to configure agent environments, install project-level instructions, author and share Agent Skills (`SKILL.md`), connect external capabilities via the Model Context Protocol (MCP), structure agent plugins, and use advanced developer extensions like Magic Context, Conductor, and the Antigravity Python SDK.

1 Configuring GitHub Copilot Settings

GitHub Copilot offers numerous configuration options that control how the AI assistant integrates into your development workflow. This section explains the key settings visible in your GitHub account preferences and provides guidance on which options to enable based on your use case.

Model Selection Options

GitHub Copilot provides access to multiple AI models, each with different capabilities and performance characteristics. The available models as of early 2026 include:

Anthropic Claude Models:

- **Claude Opus 4.1**: Most capable model for complex reasoning and analysis
  - *Pros*: Excellent at understanding nuanced requirements, handling complex codebases, superior code quality
  - *Cons*: Slower response times, may be overkill for simple tasks, limited availability (select option required)
  - *When to use*: Complex refactoring, architectural decisions, thorough code reviews
- **Claude Opus 4.5**: Latest version with enhanced capabilities
  - *Pros*: State-of-the-art performance, improved reasoning over 4.1

  - *Cons*: Similar trade-offs to Opus 4.1, requires selection
  - *When to use*: Most demanding tasks requiring cutting-edge capabilities

- **Claude Sonnet 4**: Balanced model optimizing capability and speed
  - *Pros*: Fast responses, strong performance, good default choice
  - *Cons*: Slightly less capable than Opus models for very complex tasks
  - *When to use*: General development work, most coding tasks

- **Claude Sonnet 4.5**: Enhanced version of Sonnet
  - *Pros*: Improved over Sonnet 4 while maintaining speed
  - *Cons*: Still not as powerful as Opus for extremely complex scenarios
  - *When to use*: Most daily development tasks

- **Claude Haiku 4.5**: Fast, efficient model for simpler tasks
  - *Pros*: Very fast responses, cost-effective, good for quick questions
  - *Cons*: Less capable for complex reasoning or large codebases
  - *When to use*: Simple completions, quick questions, repetitive tasks

OpenAI GPT Models:

As of 2026-05-08, OpenAI points users to ChatGPT Codex at `chatgpt.com/codex`. The OpenAI quickstart describes Codex as an AI coding assistant available through the Codex IDE extension that can read files, run commands, and write changes. This quickstart guide also links to a dedicated Codex app for working with local projects: OpenAI Codex quickstart guide. For additional background and platform context, Wikipedia describes Codex as an AI coding agent by OpenAI with desktop app availability on Windows and macOS as an additional access path: Codex (AI agent). In the GitHub Copilot model names shown below, the `-Codex` suffix identifies code-specialized variants (for example, `GPT-5.2-Codex` and `GPT-5-Codex`).

- **GPT-5.2-Codex**: Specialized for code generation
  - *Pros*: Strong code completion, good at common patterns
  - *Cons*: May hallucinate package names or APIs
  - *When to use*: Code completion, common coding patterns

- **GPT-5**: Latest general-purpose model
  - *Pros*: Broad knowledge, good general performance
  - *Cons*: Not specifically optimized for code
  - *When to use*: Mixed tasks involving code and documentation

- **GPT-5-Codex** (various versions including Mini and Max):
  - *Pros*: Specialized variants for different use cases
  - *Cons*: Fragmented options can be confusing
  - *When to use*: Specific scenarios where variant optimizations matter

Connecting Positron Assistant to OpenAI

If you are using Positron Assistant with OpenAI models, set up an OpenAI API key first.

Follow these steps:

1. Go to the OpenAI API keys page.
2. Sign in, choose or create the OpenAI project you want to use, and select **Create new secret key**.
3. Copy the key immediately and store it in a secure password manager.
4. In Positron, open the Command Palette with `Cmd+Shift+P` (or `Ctrl+Shift+P` on Windows/Linux).
5. Run `Positron Assistant: Configure Language Model Providers`.
6. Select **OpenAI**, paste your API key, and complete sign-in.

The Positron Assistant getting started guide states that OpenAI is enabled by default. If OpenAI does not appear as a provider, update Positron and confirm `positron.assistant.provider.openAI.enable` is not disabled.

Sources: Positron Assistant setup, OpenAI API key help, OpenAI quickstart.

Google Gemini Models:

- **Gemini 2.5 Pro**: High-capability model
  - *Pros*: Strong multimodal capabilities, good at understanding context
  - *Cons*: Less proven in coding scenarios than Claude or GPT
  - *When to use*: Tasks involving images or complex context
- **Gemini 3 Pro/Flash** (Preview): Latest generation
  - *Pros*: Cutting-edge capabilities, flash variant offers speed
  - *Cons*: Preview status means less stable, limited track record
  - *When to use*: Experimental workflows, evaluation of new capabilities

Lab Recommendation: For most lab work, enable **Claude Sonnet 4.5** as your default model. It provides excellent balance of capability and speed. Consider switching to **Claude Opus 4.5** for complex architectural decisions or difficult debugging sessions. Keep **Claude Haiku 4.5** enabled for quick inline completions.

Feature Settings

These settings control where and how Copilot integrates into your development environment:

Editor preview features:

- *What it does*: Enables previews of experimental features in your editor
- *Pros*: Access to latest capabilities before general release

- *Cons*: May have bugs or unstable behavior
- *Recommendation*: **Enable** if you're comfortable troubleshooting issues and want cutting-edge features

Copilot Chat in GitHub.com:

- *What it does*: Enables Copilot chat interface on GitHub.com
- *Pros*: Quick access to Copilot without opening an editor, useful for reviewing PRs
- *Cons*: Only available with paid license
- *Recommendation*: **Enable** (included in GitHub Copilot subscription)

Copilot CLI:

- *What it does*: GitHub Copilot for assistance in terminal
- *Pros*: AI help for command-line operations, shell commands, and git operations
- *Cons*: Requires separate installation and setup
- *Recommendation*: **Enable** and install via `gh extension install github/gh-copilot`

Copilot in GitHub Desktop:

- *What it does*: Enables Copilot in GitHub Desktop app
- *Pros*: AI assistance in GUI git client
- *Cons*: Limited compared to editor integration
- *Recommendation*: **Enable** if you use GitHub Desktop

Copilot Chat in the IDE:

- *What it does*: Enables chat interface in your code editor
- *Pros*: Context-aware help, refactoring assistance, code explanation
- *Cons*: Can be distracting if overused
- *Recommendation*: **Enable** (essential feature)

Copilot Chat in GitHub Mobile:

- *What it does*: Enables Copilot chat in mobile app
- *Pros*: Quick access on mobile devices
- *Cons*: Limited by mobile interface
- *Recommendation*: **Enable** for convenience

Copilot can search the web:

- *What it does*: Allows Copilot to search internet for up-to-date information
- *Pros*: Access to current documentation, recent library changes, latest best practices
- *Cons*: May introduce latency, results depend on search quality
- *Recommendation*: **Enable** for access to current information

Advanced Settings

Dashboard Entry Point:

- *What it does*: Allows instant chatting when landing on GitHub.com
- *Pros*: Quick access to Copilot without navigating menus
- *Cons*: None significant
- *Recommendation*: **Enable** for convenience

Copilot code review:

- *What it does*: Use Copilot to review your code and generate pull request summaries
- *Pros*: Automated code review suggestions, PR summary generation, catches common issues
- *Cons*: May generate false positives, shouldn't replace human review
- *Recommendation*: **Enable** (major productivity boost)

Automatic Copilot code review:

- *What it does*: Automatically reviews all pull requests you create
- *Pros*: Catches issues early without manual triggering
- *Cons*: May be noisy on simple PRs, uses API quota
- *Recommendation*: **Disable** initially; enable only after you're comfortable with code review quality

Copilot coding agent:

- *What it does*: Delegate tasks to Copilot coding agent in repositories where it is enabled
- *Pros*: Autonomous multi-file edits, can execute complex refactoring, runs tests and fixes issues
- *Cons*: Requires careful oversight, can make unwanted changes if instructions unclear
- *Recommendation*: **Enable** (see best practices for safe usage guidelines)

Copilot Memory (Preview):

- *What it does*: Remember repository context across Copilot agent interactions
- *Pros*: Better context awareness, learns repository patterns and conventions
- *Cons*: Preview feature governed by pre-release terms, potential privacy implications
- *Recommendation*: **Enable** to help Copilot learn your codebase patterns

MCP servers in Copilot:

- *What it does*: Connect MCP servers to Copilot in all editors and Coding Agent
- *Pros*: Extend Copilot with custom tools and integrations
- *Cons*: Requires MCP server setup and maintenance

- *Recommendation*: **Enable** if you have MCP servers configured; otherwise this setting has no effect

Copilot-generated commit messages:

- *What it does*: Allow Copilot to suggest commit messages when you make changes
- *Pros*: Saves time, generates descriptive messages based on code changes
- *Cons*: May miss important context, still requires review
- *Recommendation*: **Enable** but always review and edit suggested messages

Copilot Spaces:

- *What it does*: View and create Copilot Spaces (collaborative AI environments)
- *Pros*: Share AI context with team members
- *Cons*: Additional complexity for individual work
- *Recommendation*: **Enable** for team collaboration features

Copilot Spaces Individual Access:

- *What it does*: Create individually owned Copilot Spaces
- *Pros*: Personal AI workspaces for complex projects
- *Cons*: May fragment your workflow
- *Recommendation*: **Enable** for flexibility

Copilot Spaces Individual Sharing:

- *What it does*: Share individually owned Copilot Spaces
- *Pros*: Collaborate while maintaining ownership
- *Cons*: None significant
- *Recommendation*: **Enable** for sharing capability

Summary of Recommended Settings

For lab members, we recommend the following configuration:

Enable these features:

- All Copilot Chat options (GitHub.com, CLI, IDE, Mobile)
- Web search capability
- Dashboard Entry Point
- Copilot code review (but not automatic review initially)
- Copilot coding agent
- Copilot Memory
- MCP servers (if configured)
- Copilot-generated commit messages
- All Copilot Spaces options

Model selection:

- Default: Claude Sonnet 4.5
- Complex tasks: Claude Opus 4.5
- Quick completions: Claude Haiku 4.5

Enable with caution:

- Editor preview features (only if comfortable with potential instability)
- Automatic Copilot code review (wait until familiar with review quality)

Following these guidelines will help establish an effective Copilot configuration. The key is to enable features that add value to your workflow while maintaining awareness that AI assistance requires validation (see best practices).

2 Connecting VS Code to a Custom Model Endpoint (BYOK)

VS Code's built-in Chat usually talks to GitHub's hosted models. It can also route requests to a model provider of your own; GitHub calls this "bring your own key" (BYOK). The lab uses BYOK to reach Databricks model serving endpoints, which expose an OpenAI-compatible API, through the community extension `oai-compatible-copilot`.

This section describes the wiring, four errors that report themselves in the chat panel, and three more that do not.

Wiring the extension to Databricks

Databricks serves models over an OpenAI-compatible endpoint at `https://<workspace>.cloud.databricks.com/serving-endpoints`. Point the extension at it in VS Code `settings.json`:

```
"oaicopilot.baseUrl": "https://<workspace>.cloud.databricks.com/serving-endpoints",
"oaicopilot.models": [
  {
    "id": "databricks-claude-opus-5",
    "owned_by": "databricks",
    "family": "claude",
    "context_length": 64000,
    "max_tokens": 16000,
    "delay": 15000,
    "vision": true,
    "apiMode": "openai"
  },
  {
```

```
    "id": "databricks-gpt-5-4",
    "owned_by": "databricks",
    "family": "gpt-5.4",
    "context_length": 64000,
    "max_tokens": 16000,
    "delay": 15000,
    "reasoning_effort": "medium",
    "vision": true,
    "apiMode": "openai"
  },
  {
    "id": "databricks-gpt-5-3-codex",
    "owned_by": "databricks",
    "family": "gpt-5.3-codex",
    "context_length": 64000,
    "max_tokens": 16000,
    "delay": 15000,
    "reasoning_effort": "high",
    "vision": true,
    "apiMode": "openai-responses"
  }
]
```

The `id` of each model must exactly match the name of a deployed serving endpoint in your workspace. The extension sends `id` as the OpenAI `model` field, and Databricks routes the request to the endpoint of that name. Most entries use `POST /serving-endpoints/chat/completions`. Models marked as Responses-API-only in the Databricks model catalog instead require `apiMode: "openai-responses"`. That set is GPT-5.5 Pro, GPT-5.5, and GPT-5.3 Codex (measured 2026-09-09). It is not a Codex-family property: GPT-5.5 and GPT-5.5 Pro are in the set and are not Codex models. The older `gpt-5-2-codex`, `gpt-5-1-codex-max`, and `gpt-5-1-codex-mini` endpoints that earlier configurations also set to `openai-responses` no longer appear in the catalog (measured 2026-09-09). An `id` that names no real endpoint fails (see the 404 below).

To store your token, run **Set OAI Compatible Multi-Provider Apikey** from the Command Palette (`Cmd+Shift+P` / `Ctrl+Shift+P`), choose the `databricks` provider, and paste a Databricks personal access token. The extension keeps it in VS Code's encrypted secret storage (under `oaicopilot.apiKey.databricks`) and sends it as an `Authorization: Bearer` header.

Sizing context and output limits

The model metadata controls both the request and the amount of conversation history Copilot sends:

- `context_length` is the total context window that the extension advertises to Copilot. It may deliberately be smaller than the provider's maximum window.
- `max_tokens` is the output cap. The extension maps it to `max_output_tokens` in Responses mode.
- `family` selects the closest Copilot system-prompt family.
- `vision` tells Copilot whether it may send images.

The extension advertises input capacity as `context_length` minus `max_tokens`. For example, `context_length: 64000` with `max_tokens: 16000` allows Copilot to send about 48,000 input tokens. Use the full provider window only when the workspace quota can sustain repeated agent turns at that size.

The underlying model's maximum output is not always a good value for `max_tokens`. Databricks reserves the requested output allowance before admitting a request. Under the standard pay-per-token quota, Claude, GPT-5, and Gemini models generally have a 20,000 output-token-per-minute limit. A 64,000-token request can therefore receive an immediate 429 response even when the underlying model supports that output length.

Use the quota-aware lab defaults in Table 1. The context column is the operational value to put in `settings.json`, not the model's maximum capability.

Table 1: Lab defaults for Databricks-hosted models

Model group	Workspace ITPM / OTPM	OAICopilot context	Output cap	Delay
GPT-5.6 Sol/Terra/Luna	2,000,000 / 200,000	400,000	16,000	0 ms
Claude Opus/Sonnet/Haiku	200,000 / 20,000	64,000	16,000	15,000 ms
GPT-6 Astra and GPT-5.5 through GPT-5	200,000 / 20,000	64,000	16,000	15,000 ms
Gemini	200,000 / 20,000	64,000	16,000	15,000 ms
GLM 5.3/5.3 Flash/5.2	200,000 / 20,000	64,000	16,000	15,000 ms
Grok 4.6	200,000 / 20,000	64,000	16,000	15,000 ms
Inkling	200,000 / 10,000	56,000	8,192	15,000 ms
Kimi K3	200,000 / 10,000	56,000	8,192	15,000 ms
DeepSeek V4 Flash	200,000 / 10,000	56,000	8,192	15,000 ms
DeepSeek V4 Pro	200,000 / 4,000	64,000	2,000	30,000 ms

Model group	Workspace ITPM / OTPM	OAICopilot context	Output cap	Delay
GPT OSS 120B/20B	1,000,000 / 100,000	131,072	25,000	0 ms
Qwen3.5 122B	1,000,000 / 100,000	131,072	25,000	0 ms
Llama 4 Maverick	1,000,000 / 100,000	128,000	8,192	0 ms
Llama 3.3/3.1 and Gemma 3	1,000,000 / 100,000	128,000	8,192	0 ms

The ITPM / OTPM column is the Enterprise-tier pay-per-token limit from the Databricks limits page (Databricks 2026a), and the endpoint names are from the model catalog (Databricks 2026b), both measured 2026-09-09. The endpoint ids for the rows added on that date are `databricks-gpt-6-astra`, `databricks-glm-5-3`, `databricks-glm-5-3-flash`, `databricks-glm-5-2`, `databricks-grok-4-6`, `databricks-kimi-k3`, `databricks-deepseek-v4-flash-0731`, `databricks-deepseek-v4-pro-0813`, and `databricks-qwen35-122b-a10b`. The context column is a quota-derived working value, not a provider window. The catalog gives 1 million tokens of context for GLM, Kimi K3, and Inkling, and no window at all for either DeepSeek endpoint (measured 2026-09-09); on a 200,000 ITPM tier a 1 million-token `context_length` overruns every budget in Table 2. Qwen3.5's 256K window and 25K output cap (measured 2026-09-09) sit on the 1,000,000 tier, so its row copies the GPT OSS values, and its 25,000 output cap is the provider figure, which that tier's 100,000 OTPM admits. `databricks-inkling` is a real catalog endpoint, listed under Thinking Machine Labs as a Public Preview model that always reasons (measured 2026-09-09), so its row documents a live deployment rather than a local alias. The output cap is a per-request ceiling that most turns never reach, which is why the 20,000 OTPM rows carry a 16,000 cap at four requests per minute. A 4,000 OTPM tier has no such slack: DeepSeek V4 Pro's 2,000 cap and 30,000 ms delay keep two requests inside the allowance, and 2,000 tokens is too little for an agent-mode edit, so treat that row as a chat-only default and prefer DeepSeek V4 Flash for agent work.

Use `openai-responses` for the endpoints named in the wiring section above, and `openai` for the rest. Read the catalog rather than that list, since an endpoint can change API mode without any model being added. Add the `delay` value to each affected model entry. The extension applies this model-specific pause between requests; models without it fall back to the global `oaicopilot.delay` value. Every context and output cap in the table is a working default, not the underlying model's maximum capability. Workspaces with higher provisioned or priority limits can raise these after checking their actual quota.

Why the context and delay columns move together

The context and delay columns are one setting expressed twice, so changing either alone breaks the pairing.

A workspace admits `ITPM` input tokens per minute. The extension advertises `context_length` minus `max_tokens` as the model's input budget, and the delay sets how often a request can be sent. Sustained work therefore needs

$$\text{input budget} \times \text{requests per minute} \leq \text{ITPM}$$

The Claude row is tuned to sit just under that ceiling. A 15,000 ms delay allows four requests per minute, and each request carries the 48,000-token input budget derived above, so a busy client draws about 192,000 against a 200,000 ITPM tier. That is roughly 96 percent of the allowance, which is why the 64,000 value looks conservative and is not.

This is what makes a larger window expensive. Raising `context_length` while leaving the delay alone multiplies straight through the inequality above. Table 2 gives the largest input budget each tier sustains at a given pace.

Table 2: Largest sustainable input budget by tier and pacing

Workspace ITPM	15 s (4/min)	30 s (2/min)	60 s (1/min)
200,000	50,000	100,000	200,000
1,000,000	250,000	500,000	1,000,000
2,000,000	500,000	1,000,000	2,000,000

A model's own maximum window is a separate quantity from either column, and it is usually far larger. Registering it directly is the common mistake. Advertising a 1,000,000-token window on a 200,000 ITPM tier offers a single prompt of 984,000 input tokens, which is 4.9 times the entire per-minute allowance, so two full prompts would need about 295 seconds between them. The window is a real capability of the model and it is not available at that quota.

Choosing a model for agent mode

Agent mode spends the input budget faster than chat does, because tool definitions and file contents are sent before any conversation. Two configurations are possible, and the tier decides which one is open.

On a 200,000 ITPM tier the budget can be spent on a large window or on frequent turns, and not on both. Doubling the window means halving the pace, so an agent that reads several files per turn slows to a crawl exactly when it is doing the most work.

The higher tiers change the answer rather than easing it. GPT-5.6 Sol, Terra, and Luna sit on a 2,000,000 ITPM tier, ten times the Claude and Gemini families, and carry a 400,000 context

with no delay. That is 384,000 input tokens at roughly 5 requests per minute: 8 times the Claude window and faster turns at the same time. Prefer that family for agent work on a standard pay-per-token workspace, and keep the 200,000 ITPM families for chat.

Note the direction of the trade, which is the opposite of the intuition. Within one tier a smaller window buys more turns per minute, so the fastest agent configuration is rarely the widest one.

Measure before tuning either column. VS Code's status bar reports tokens used against the advertised window for the current chat, which is the only reading that reflects what a client actually sends. Read it in a **new** chat, since an existing one keeps the context metadata it was created with.

The 64,000/16,000 combination limits one prompt to about 48,000 input tokens. Together with 15-second pacing, that keeps one busy client near rather than far above a 200,000 ITPM tier. It is not a guarantee: the quota is shared across the workspace, prompt sizes vary, and concurrent users or clients consume the same allowance. Increase the delay or reduce `context_length` further when 429s continue.

Use longer retry spacing than the extension's one-second default:

```
"oaicopilot.retry": {
  "enabled": true,
  "max_attempts": 3,
  "interval_ms": 15000,
  "status_codes": []
}
```

The extension already retries 429 responses and doubles this base interval on successive attempts. The longer starting interval gives Databricks' sliding token window time to recover.

For GPT-5 and GPT OSS models, `reasoning_effort` adds a selector to the Copilot model configuration and is forwarded to Databricks. Start with `medium` for general work and `high` for Codex or difficult agentic tasks. Claude and Gemini 2.5 use provider-specific thinking controls; do not copy `reasoning_effort` onto those entries.

Databricks retires model endpoints over time. Before sharing or troubleshooting a configuration, compare every `id` with the current model catalog and remove entries that are no longer listed. An accurate local entry cannot make a retired endpoint work.

An individual endpoint page may also carry a banner reading "This serving endpoint is deprecated. Foundation models are now managed in Unity AI Gateway." That is a statement about where Databricks intends to manage these models, not an outage: the `/serving-endpoints` path keeps serving while the banner is up. Unity AI Gateway's **LLMs** and **Providers** tabs are in beta, and they are inactive in a workspace that has not been enabled for them. In that state there is no gateway base URL to move to, and the configuration

in this section is still the working one. Re-check when those tabs become active. Observed 2026-08-20.

Four errors that report themselves, and why they stack

These failures sit on top of each other: fixing one uncovers the next, so work through them top-down.

1. "No utility model is configured for 'copilot-utility-small'"

When your main Chat model is a BYOK model, VS Code still needs a small "utility" model for background chores such as generating the conversation title and naming git branches. If none is configured, Chat fails before it ever reaches your provider:

```
No utility model is configured for 'copilot-utility-small'
while the selected main agent model is BYOK.
```

Set `chat.byokUtilityModelDefault` in `settings.json`:

- `"mainAgent"`: reuse your BYOK main model for these chores. This keeps all traffic on your provider and needs no extra endpoint, so it is the simplest choice.
- `"copilot"`: use GitHub's hosted utility model. This needs an active Copilot subscription.
- `"none"`: the default, which errors on purpose.

This requirement arrived in a mid-2026 VS Code update. Before that, BYOK chat worked without the setting, so an editor update can make a working setup start failing here.

2. `[404] ENDPOINT_NOT_FOUND`

```
[404] Not Found
{"error_code":"ENDPOINT_NOT_FOUND",
 "message":"The given endpoint does not exist, please retry after
            checking the specified model and version deployment exists."}
```

The `model` name in the request is not a serving endpoint that exists in the workspace. Check that every `id` in `oaicopilot.models` matches a real, deployed endpoint (Databricks workspace → **Serving**), and remove or rename any entry that points at a name with no deployment. A stray placeholder entry, such as a leftover `copilot-utility-small`, is a common cause.

3. `[403] Invalid access token`

```
[403] Forbidden
{"error_code":403,"message":"Invalid access token."}
```

The stored token is expired or revoked. Databricks OAuth tokens are short-lived and can expire within the day, so a session that worked in the morning can start returning 403 by afternoon; personal access tokens last until their configured expiry. Generate a fresh token (Databricks → **Settings** → **Developer** → **Access tokens**) and re-run **Set OAI Compatible Multi-Provider Apikey**. Prefer a long-lived personal access token to avoid frequent re-authentication. No window reload is needed; the extension reads the token on each request.

4. `[429] REQUEST_LIMIT_EXCEEDED`

```
[429] Too Many Requests
REQUEST_LIMIT_EXCEEDED: Exceeded workspace input tokens per minute rate limit
for databricks-claude-sonnet-5.
```

This example is an input-tokens-per-minute (ITPM) failure. Lowering only `max_tokens` does not fix it: Databricks counts the actual prompt and conversation history against ITPM, while `max_tokens` reserves output-tokens-per-minute (OTPM) capacity.

For an ITPM error:

1. lower the model's operational `context_length`;
2. add or increase its model-specific `delay`;
3. lengthen `oaicopilot.retry.interval_ms`;
4. start a new conversation when accumulated history is no longer useful;
5. use GitHub's utility model for background chores when your Copilot plan allows it; and
6. ask the Databricks account team for a higher tier, or use provisioned throughput for sustained workloads.

For an OTPM error, lower `max_tokens` first. For either type, the error can persist until the sliding rate-limit window recovers.

> **Tip**
>
> A quick way to tell 404 from 403: a 404 means the request authenticated but named a missing endpoint (a model-name or configuration problem), while a 403 usually means the token itself was rejected (an authentication problem). The IP access-list 403 below is the exception — there the token is valid and the failure is at the network boundary.

> **Check the workspace host before minting another token**
>
> Both the 404 and the 403 above assume the request reached the workspace you think it did. A `baseUrl` whose `dbc-` host belongs to a *different* workspace produces those same two errors and survives every remedy listed for them. The endpoint name is real and the token is valid; neither one is in the workspace being asked. Issuing a fresh token then fails in exactly the same way, indefinitely.

Compare the host in `oaicopilot.baseUrl` against the invocations URL shown at the top of the endpoint's page in the Databricks console (**Serving**, then the endpoint). Check every per-model `baseUrl` as well: each model entry may carry its own copy of the host, so a single corrected setting can leave dozens of stale ones behind it.
Observed 2026-08-20, where a stale host appeared 42 times in one `settings.json`: once at the top level and once in each of 41 model entries. A second VS Code installation on the same machine carried the same stale host in its own copy of the setting.

! A 403 that no new token can fix — IP access list

There is a third 403 with the same status line but a different message body and a different remedy:

```
[403] Forbidden
{"error_code":403,"message":"Source IP address: <ip> is blocked by Databricks IP ACL for wo
id> [ReqId: ...]"}
```

The request left from an address outside the workspace's IP access list, typically because a VPN dropped or was never connected. The token is valid, the host is correct, and the endpoint exists, so the request succeeds in intent and fails at the network boundary. No credential change resolves it — minting a fresh token repeats the same failure indefinitely, the same shape as the stale-host case one layer further out.
Connect to the network that the workspace allows (restore the VPN or move to an allowed address) and retry; the existing token will then succeed without replacement.
The IP-ACL variant is distinguishable by its message body, which names your source IP and a workspace id. The other two 403 situations are not distinguishable by body alone — both the expired-token case and the wrong-workspace case described in the preceding callout surface as `Invalid access token` (and, for a missing endpoint, as `ENDPOINT_NOT_FOUND`):

Message body	Cause	Remedy
`Invalid access token`	expired or revoked token, or `baseUrl` names the wrong workspace	mint a new token or correct the host (see preceding callout)
`Source IP address: ... blocked by Databricks IP ACL for workspace: <workspace-id>`	off-network / VPN down	connect to the allowed network; no credential change

Observed 2026-08-26 on `databricks-gpt-5-6-sol` against `dbc-440c7148-9ff6`, three consecutive requests while a VPN connection was down.

Three failures that name no error in the chat panel

The four errors above print their own text. These three do not name an error. Failures 5 and 6 leave an ordinary-looking reply in the chat panel, and the only record is in VS Code's **GitHub Copilot Chat** output channel (**View**, then **Output**, then pick that channel). Failure 7 is visible in the panel as a `[object Object]` prefix on the reply; it is still a display bug rather than a named error. Open the output channel first whenever a BYOK reply is wrong in a way that names no error.

A reply that begins `[object Object]` and then answers as though you had asked nothing is the symptom that produced all three of these at once. Observed 2026-08-20 with VS Code 1.135.0-insider, Copilot Chat 0.63.2026082004, `oai-compatible-copilot` 0.4.2, and `databricks-claude-sonnet-5`.

5. `OAI Compatible API key not found`

```
OAI Compatible API key not found
  at ...provideLanguageModelChatResponse (out/provider.js:173)
```

The extension keeps the token in VS Code's encrypted secret storage, which is per install, not per profile. `settings.json` travels through Settings Sync; the secret does not. So a second install, such as Insiders beside stable, shows a complete-looking `oaicopilot` configuration with no token behind it. A new profile in the same install still sees the existing token. Re-run **Set OAI Compatible Multi-Provider Apikey** in the install that is failing.

This is not the same as the 403 above. There, a token was sent and the provider rejected it; here, no request reaches the provider at all.

6. `No lowest priority node found`

```
Error: No lowest priority node found (path: ...)
  ... [ConversationHistorySummarizer] summarization failed
```

This one comes from Copilot Chat's prompt renderer, not from Databricks. The renderer drops prompt elements in priority order until the prompt fits the input budget the extension advertises, which is `context_length` minus `max_tokens`, or 48,000 tokens at the Claude defaults in Table 1. The error is what it raises when it has nothing left to drop and the prompt is still over budget. A prompt pruned that far need not still contain your own message, which fits a model that replies it cannot see a request.

Agent mode reaches this sooner than ordinary chat, because tool definitions and instruction files consume the budget before any conversation does. Raise the model's `context_length` and leave `max_tokens` alone: that widens the input allowance without reserving more output tokens per minute. Reducing the number of active tools and instruction files works too. Weigh both against Table 1, whose context values are chosen to keep one client inside an ITPM

tier, so buying prompt headroom this way costs more 429s. Raising the window without lengthening the delay breaks the pairing those two columns encode, and on a 200,000 ITPM tier the headroom is not there to buy. Switching to a higher-tier family is the move that gets both, as Table 2 sets out.

7. `[object Object]` in the reply text

Version 0.4.2 renders streamed content with `String(deltaObj.content)` in both of its streaming paths, in `out/openai/openaiApi.js`. A chunk whose `content` is a plain string renders normally. A chunk that carries structured content instead prints as the literal text `[object Object]`, because that is what JavaScript's `String()` returns for an object. Later chunks that carry a plain string render normally. In the 2026-08-20 session the prefix appeared once, at the start of the reply. That is an observation from that session, not a guarantee that later chunks cannot also be structured.

This is a display bug in the extension rather than a configuration error, so no setting turns it off. Report it upstream and read past the prefix.

> **Note**
>
> Copilot's own hosted quota still applies to some background chores even when the main chat model is BYOK. A log line reading `quotaExceeded | gpt-4o-mini-2024-07-18 | [title]` is conversation-title generation failing against GitHub's models, and it says nothing about whether your provider is working. In the 2026-08-20 session above, `chat.byokUtilityModelDefault` was already set to `"mainAgent"` and the title request still went to `gpt-4o-mini`, so that setting did not cover conversation titles in this version.

3 Configuring the Agent Environment

The `.github/workflows/copilot-setup-steps.yml` file allows you to customize the development environment in which the GitHub Copilot coding agent operates. This file preinstalls tools and dependencies so that Copilot can build, test, and lint your code more reliably.

Why Configure the Environment?

While Copilot can discover and install dependencies through trial and error, this can be slow and unreliable. Additionally, Copilot may be unable to access private dependencies. Preconfiguring the environment ensures:

- Faster agent startup and execution
- More reliable builds and tests
- Access to private or authenticated dependencies
- Consistent development environment across all agent sessions

File Location and Structure

The workflow file must be located at `.github/workflows/copilot-setup-steps.yml` in your repository's **default branch**. It follows GitHub Actions workflow syntax but must contain a single job named `copilot-setup-steps`.

Basic Configuration Example

See this repository's own `.github/workflows/copilot-setup-steps.yml` for a configuration adapted for R and Quarto projects.

Using `actions/checkout`

The `actions/checkout` action is used to check out your repository code so that the workflow can access it. While Copilot will automatically check out your repository if you don't include this step, **explicitly including it is necessary** when your setup steps need to access repository files.

Why explicitly include checkout?

Many dependency installation steps require access to repository files:

- `r-lib/actions/setup-renv@v2` needs `renv.lock` to install R package dependencies
- `r-lib/actions/setup-r-dependencies@v2` needs `DESCRIPTION` to install R package dependencies
- `npm ci` needs `package-lock.json` to install Node.js dependencies
- `pip install -r requirements.txt` needs the requirements file

Without an explicit checkout step, these dependency installation commands will fail because the necessary files won't be available yet.

Basic checkout:

```yaml
- name: Checkout code
  uses: actions/checkout@v4
```

Important: The Copilot coding agent overrides any `fetch-depth` value you set in the checkout step. According to GitHub's official documentation, this override happens "to allow the agent to rollback commits upon request, while mitigating security risks." The agent dynamically determines the appropriate fetch depth based on the pull request context.

While you cannot control the fetch depth used by Copilot, the agent still has access to sufficient git history to perform its work effectively, including comparing changes and understanding the context of your pull request.

Configurable Options

You can customize only these specific settings in the `copilot-setup-steps` job:

- `steps`: Setup commands and actions to run
- `permissions`: Access permissions (typically `contents: read`)
- `runs-on`: Runner type (Ubuntu x64 Linux only)
- `services`: Database or service containers
- `snapshot`: Save environment state
- `timeout-minutes`: Maximum 59 minutes

All other workflow settings are ignored by Copilot.

Common Setup Tasks

For Node.js/TypeScript projects:

```yaml
- name: Set up Node.js
  uses: actions/setup-node@v4
  with:
    node-version: "20"
    cache: "npm"

- name: Install dependencies
  run: npm ci
```

For Python projects:

```yaml
- name: Set up Python
  uses: actions/setup-python@v5
  with:
    python-version: "3.11"

- name: Install dependencies
  run: pip install -r requirements.txt
```

For R projects:

```yaml
- name: Set up R
  uses: r-lib/actions/setup-r@v2
  with:
    r-version: 'release'
```

```yaml
- name: Install R dependencies
  uses: r-lib/actions/setup-renv@v2
```

Environment Variables and Secrets

To set environment variables for Copilot:

1. Navigate to your repository's **Settings**
2. Go to **Environments**
3. Select or create the `copilot` environment
4. Add environment variables or secrets as needed

Use secrets for sensitive values like API keys or passwords.

Testing Your Configuration

The workflow runs automatically when you modify `copilot-setup-steps.yml`, allowing you to validate changes in pull requests. You can also manually trigger the workflow from the repository's **Actions** tab.

Setup logs appear in the agent session logs when Copilot starts working. If a step fails, Copilot will skip remaining steps and begin working with the current environment state.

Advanced Configuration

Larger runners: For projects requiring more resources, you can use larger GitHub-hosted runners:

```yaml
jobs:
  copilot-setup-steps:
    runs-on: ubuntu-4-core
```

Self-hosted runners (ARC): For access to internal resources or private registries, use Actions Runner Controller (ARC) self-hosted runners:

```yaml
jobs:
  copilot-setup-steps:
    runs-on: arc-scale-set-name
```

Note: When using self-hosted runners, you must disable Copilot's integrated firewall in repository settings and configure appropriate network security controls.

Git Large File Storage (LFS): If your repository uses Git LFS:

```yaml
- uses: actions/checkout@v4
  with:
    lfs: true
```

Further Reading

For complete details, see Customizing the development environment for GitHub Copilot coding agent.

4 Agent Sessions and Handoff in Visual Studio Code

In Visual Studio Code, interactions with AI coding assistants are structured around Agent Sessions and Handoff (measured 2026-08-31). Understanding how sessions organize work and transfer state across tools is essential for managing multi-step agent workflows.

Anatomy of an agent session

An agent session represents a stateful stream of interaction between a developer and an agent. Each session maintains:

- **Chats and model selection**: One or more chats within a session, each with its own agent or model selection.
- **Interaction history**: User prompts, assistant responses, and tool calls executed during the task.
- **Accumulated context**: Referenced files, code snippets, and conversational state gathered across turns.
- **Checkpoints and branching**: Points in session history allowing developers to fork or roll back state when exploring alternative implementation paths.

Managing sessions across surfaces

VS Code provides unified session discovery and access across editor surfaces:

- **Shared surfaces**: The primary Chat view and the dedicated Agents window share the same sessions, allowing developers to switch views without losing conversational context.
- **External session discovery**: VS Code discovers sessions initiated outside the primary GUI, including CLI agent sessions (such as Copilot CLI, Claude Code, and Codex).
- **Cloud synchronization**: Synced sessions are backed up to your GitHub account, enabling developers to access active and past sessions across devices.

Session handoff types

Session handoff transfers context and intent from an active session to a specialized workflow without manual re-prompting:

- **Harness to harness**: Switch the active session between different agent harnesses to leverage distinct agent runtime capabilities on the same task.
- **Plan to implementation**: Hand off a high-level architectural plan or task specification directly to an implementation session to generate code.
- **Continue in the cloud**: Hand off a local session to run in a cloud-hosted agent environment (such as background tasks leading to pull requests), freeing local editor resources while the agent executes in the background.

5 Installing Claude Code on Windows

Claude Code is Anthropic's command-line coding agent. Installing it on Windows works well, but a few platform-specific pitfalls can cost you hours if you don't know about them. These notes capture a setup that works, and the gotchas to watch for.

> **Note**
>
> These notes were written in June 2026. As with the rest of this chapter, treat the specifics with caution— installers and behavior change quickly.

Run it from a Unix-like terminal

Claude Code expects a Unix-like shell. On Windows, run it from one of:

- **Git Bash** (ships with Git for Windows);
- **MSYS2** (https://www.msys2.org/), which also gives you a package manager (`pacman`) and optional `zsh`;
- **WSL** (Windows Subsystem for Linux), the most Unix-faithful option.

If you use WSL, install Claude Code *inside* WSL— a Windows install will not carry over, because WSL has its own filesystem and `PATH`. Inside WSL the standard Linux install applies, and the Windows-specific `PATH` and `rehash` gotchas below don't apply.

Install Claude Code

There are two common routes:

1. **Native installer** (recommended):

```
curl -fsSL https://claude.ai/install.sh | bash
```

   This installs the binary to `~/.local/bin`. If your organization's policy requires it, download and inspect the script before running it rather than piping it straight to `bash`.

2. **npm** (requires Node.js):

```
npm install -g @anthropic-ai/claude-code
```

> **!** Claude Code migrates itself out of npm
>
> Even if you install via npm, current versions **migrate themselves to a native install** the first time you run `claude` (at `~/.local/bin/claude`, or `~/.local/bin/claude.exe` on Windows) and remove the npm copy. You can watch this happen in the terminal output the first time you run `claude`; the current install methods are documented in the Claude Code setup guide.
> This is the single most confusing Windows gotcha: a path that worked a moment ago "disappears." **Do not** hardcode the npm location (e.g. `.../AppData/Roaming/npm/...`) in your shell config. Point your `PATH` at the shell-appropriate directory shown in the next section instead.

Make sure your shell can find `claude`

The native binary lives in `~/.local/bin`. The installer adds this directory to `PATH` for ordinary shells, but on Windows two things commonly break that.

MSYS2 does not inherit the Windows `PATH` by default. It starts in a "minimal" path mode, so tools installed elsewhere on Windows are invisible to it. Add the binary's directory explicitly in your `~/.zshrc` (or `~/.bashrc`). Because MSYS2's `$HOME` is `/home/<you>` (its own home, not the Windows profile where the installer puts the binary), point `PATH` at the absolute Windows path:

```
export PATH="/c/Users/<you>/.local/bin:$PATH"     # MSYS2
```

In **Git Bash**, `$HOME` already is `/c/Users/<you>` (your Windows user profile), so the shorthand works there:

```
export PATH="$HOME/.local/bin:$PATH"     # Git Bash
```

Rehash after changing `PATH`. `zsh` and `bash` cache the locations of executables. If you add a directory to `PATH` in your shell config, the shell may still report `command not found` *even*

though the directory is on `PATH`, because its command table is stale. Force a rebuild in the same shell right after editing `PATH`:

```
rehash        # zsh; use 'hash -r' in bash
```

This bites hardest with oh-my-zsh, which builds zsh's command table *before* your custom `PATH` line runs, so opening a fresh window may not clear the stale `command not found` on its own until you `rehash` (or move the `PATH` line before oh-my-zsh initializes).

Do not edit dotfiles with PowerShell redirection

> ⚠ PowerShell writes the wrong encoding
>
> Never write to `~/.bashrc`, `~/.zshrc`, or other shell config files using PowerShell's `>` or `>>` redirection. Windows PowerShell writes **UTF-16 with a byte-order mark**, which `bash`/`zsh` read as a stray character at the start of the file:
>
> ```
> bash: $'\377\376export': command not found
> ```
>
> Edit dotfiles from *inside* the shell (e.g. with `nano`, `vim`, or `echo 'export PATH=...' >> ~/.zshrc` run in bash/zsh), or with an editor that saves UTF-8 without a BOM (e.g. VS Code).

Authenticating the GitHub CLI in Git Bash or MSYS2

If you also use the GitHub CLI (`gh`) to push code or open pull requests, `gh auth login` may fail with:

```
could not prompt: … running in MinTTY without pseudo terminal support
```

Git Bash and MSYS2 both default to the MinTTY terminal, which can't host the interactive prompt. Wrap the command with `winpty`, or run it from PowerShell / Windows Terminal instead:

```
winpty gh auth login
```

`winpty` ships with Git Bash, but in MSYS2 it's a separate package — install it first with `pacman -S winpty` if you get `command not found`.

Verify the install

Open a **new** terminal window (so it picks up your updated config) and run:

```
claude --version      # prints the installed version number
```

If you get a version number, you're ready to run `claude` in your project directory. If you get `command not found`, re-check the two `PATH` issues above: the directory must be on `PATH`, and you must `rehash` (or open a fresh window) after changing it.

6 Herdr: A Terminal Multiplexer for Coding Agents

Herdr (herdrdev 2026e) is a terminal multiplexer built for running several coding agents at once (measured 2026-09-09). It describes itself as "the runtime your coding agents live on": a background server owns the real terminal processes, clients attach to render them, and the server keeps every agent running when the client closes or an SSH connection drops. On top of that `tmux`-like core it detects which agent is running in each pane and reports whether that agent is working, blocked, done, or idle, so the operator looks only at the pane that needs attention.

Maker and activity

Herdr is developed by herdrdev (herdrdev 2026f), a one-person company founded in 2026 by Can Celik in Ankara, Turkey, and part of the Y Combinator Fall 2026 batch (Y Combinator 2026). The project is written in Rust and licensed under Apache 2.0. The GitHub repository was created on 2026-03-27 and had about 37,000 stars, 2,700 forks, and 326 open issues on 2026-09-09, with the most recent commit the day before; the latest stable release was v0.9.0 on 2026-09-07, following v0.8.2 (2026-08-19) and v0.8.0 (2026-08-03) (herdrdev 2026f). A Show HN thread drew 166 points and 110 comments (Hacker News 2026), and a community-maintained list of plugins and integrations has grown around the tool (Konur 2026). The software is pre-1.0 and changes quickly, so expect some of what follows to be out of date.

What it does

Herdr organizes terminals into a hierarchy of sessions, workspaces, tabs, and panes (herdrdev 2026b). A pane is a real terminal (a PTY) that the server owns; a workspace groups the tabs and panes for one repository, task, or investigation. The distinctive features, beyond ordinary multiplexing, are:

- **Agent status detection**: Herdr recognizes 21 coding agents out of the box, including Claude Code, Codex, Cursor Agent CLI, OpenCode, GitHub Copilot CLI, Hermes Agent, Pi, Amp, Grok CLI, Antigravity CLI, and Kiro CLI, with Gemini CLI and Cline detected but less thoroughly tested (herdrdev 2026a). Where an agent ships lifecycle hooks, the hook reports state authoritatively; otherwise Herdr matches the bottom of the live terminal buffer against a TOML detection manifest, and marks a pane `blocked` only when that snapshot matches a known approval, question, or permission prompt.
- **Git worktree management**: the socket API exposes `worktree.list`, `worktree.create`, `worktree.open`, and `worktree.remove`, which "manage Git checkouts as Herdr workspaces" (herdrdev 2026i). Creating a worktree checks out (or creates) a branch and opens the checkout as a new workspace grouped under the source workspace; the CLI form is `herdr worktree create --branch <name>` (Copes 2026).
- **A CLI and socket API for agents to drive**: the two are "the same surface agents drive", speaking newline-delimited JSON over a Unix socket (a named pipe on Windows) at `~/.config/herdr/herdr.sock`. Methods include `pane.read` to read a pane's recent output, `agent.prompt` to send a prompt, and `agent.wait` to block until another agent reaches `done` or `blocked` rather than polling (herdrdev 2026i).
- **Persistence and resume**: detaching with `ctrl+b q` leaves every process running; a server restart stops processes but restores workspaces, tabs, panes, working directories, and layout from a `session.json` snapshot, and supported agents can resume their own conversation (for example `claude --resume <id>`) from a saved session reference (herdrdev 2026h).
- **Remote machines**: `herdr machine add <ssh-host>` registers a machine reachable over ordinary SSH, installs or starts the Herdr server there, and folds its agents into one combined agent list; authentication stays with OpenSSH, and Herdr stores no keys or passwords (herdrdev 2026c). Remote servers must run Linux or macOS; a native Windows server is not supported as an SSH target.

Install and use

Herdr ships stable binaries for Linux, macOS, and Windows x86_64 (herdrdev 2026g). The one-line installers are:

```
curl -fsSL https://herdr.dev/install.sh | sh
```

```
powershell -ExecutionPolicy Bypass -c "irm https://herdr.dev/install.ps1 | iex"
```

Homebrew (`brew install herdr`), `mise`, and Nix are also supported. Run `herdr` to start the server and attach a client, `herdr agent list` to see detected agents and their states, `herdr status` to summarize the runtime, and `herdr update` to upgrade. The default prefix key is `ctrl+b`, as in `tmux`, and the configuration file lives at `~/.config/herdr/config.toml` (`%APPDATA%\herdr\config.toml` on Windows) (herdrdev 2026d). The maintainers publish an

agent guide (herdrdev 2026d) meant to be handed to a coding agent so it can set Herdr up for you.

Pricing and license

The runtime is free and open source under the Apache 2.0 license. The landing page announces a paid "Herdr Cloud" as "coming soon", described as connecting your own machines without configuring SSH, with a waitlist and no published pricing (measured 2026-09-09) (herdrdev 2026e).

Useful to us?

Herdr overlaps with two things the lab already does (our orchestration baseline): running several Claude Code sessions in parallel, and isolating each one in its own git worktree. It does not replace either. Herdr runs the same agent CLIs unchanged, and its worktree commands wrap the same `git worktree` calls that Claude Code's own worktree isolation makes. What it adds is the layer between those sessions and the person watching them:

- **An attention queue instead of a wall of terminals.** When a dozen sessions run at once, the expensive part is finding the one waiting on a permission prompt. The blocked, working, and done states Herdr aggregates across every workspace and every machine answer that question at a glance, which is the gap one early adopter reported it filling when herding parallel agents on a remote box (Coles 2026).
- **Sessions that survive a closed laptop.** Long runs of the lab's review-and-iterate loop currently die with the terminal that started them unless they are already running on a server under `tmux`. Herdr gives the same persistence with a friendlier client, and can reattach to a Claude Code conversation by session ID after a restart.
- **A machine-readable control surface.** `agent.wait` and `pane.read` let a supervising agent spawn helpers in separate worktrees, wait until each is done or blocked, and read their output, without the polling loops our current scheduled check-ins use.

The same early adopters name the limits (Coles 2026; Copes 2026). Herdr provides no sandboxing and does not isolate file changes between agents sharing a directory; worktrees, permissions, and network egress remain the user's problem, so our hooks and permission rules stay in place unchanged. Windows support is in beta, a Windows machine cannot serve as a remote target, and the tool is pre-1.0. It also sits beside, not inside, the editor-hosted session views described in Section 4: VS Code discovers CLI sessions and hands work between harnesses, while Herdr owns the terminals those sessions run in.

The practical recommendation is to try Herdr on a Linux workstation or server where a lab member already runs several Claude Code sessions, using one workspace per worktree. It is a small, reversible addition (no wrapper around the agent, no change to `ai-config`), and the status sidebar alone may justify it. Adopting the socket API as an orchestration layer is a larger step that we should defer until the project reaches a stable release.

7 Customizing an Agent

Section 9 describes one way to extend an agent. It is not the only one, and a lab that knows only that one tends to write every customization as a skill, including the ones that should have been something else.

This section maps the whole surface. The mechanisms differ less in what you can write in them — most are Markdown with a YAML front matter header, as harness construction describes — than in **when they fire and who decides**.

The Question That Picks the Mechanism

Ask two things about the behavior you want:

- **Who triggers it?** You, by typing something; the model, by judging it relevant; or the harness, on a fixed event.
- **What happens if the model disagrees?** Some mechanisms are advice the model may ignore. Others are enforced by the harness whatever the model decides.

That second question is the one people get wrong, and Claude Code's own documentation is blunt about it. Instruction files are described as "context, not enforced configuration", delivered "as a user message after the system prompt", so "there's no guarantee of strict compliance." The same page names the remedy:

> To block an action regardless of what Claude decides, use a PreToolUse hook instead.

A rule you cannot afford to have ignored does not belong in a `CLAUDE.md`.

Instruction Files: Always-On Context

`CLAUDE.md` and `AGENTS.md` are prose the harness loads at the start of a session, with no front matter and no schema (harness construction).

Three properties matter when you write one:

- **Files concatenate; they do not override.** Claude Code walks up the directory tree from the working directory, and "all discovered files are concatenated into context rather than overriding each other", ordered from the filesystem root down. A project file does not replace your personal one.
- **Imports exist, and they skip code spans.** The `@path/to/import` syntax pulls in another file, recursively, to a maximum depth of four hops. Paths resolve relative to the importing file. To *mention* a path without importing it, wrap it in backticks — import parsing skips fenced code blocks and code spans.

- **The two filenames are not interchangeable.** Claude Code reads `CLAUDE.md`, not `AGENTS.md`, and recommends a `CLAUDE.md` whose first line is `@AGENTS.md` so both tools read one source. A symlink works on macOS and Linux; on Windows it needs Administrator privileges or Developer Mode, so the import is the portable choice. GitHub Copilot, by contrast, reads all of `AGENTS.md`, `CLAUDE.md`, and `GEMINI.md`.

`AGENTS.md` is stewarded by the Agentic AI Foundation under the Linux Foundation, and it is worth being precise about what it standardizes: a filename, a location, and a nearest-file-wins precedence rule. Asked whether there are any required fields, its own FAQ answers that there are none — "AGENTS.md is just standard Markdown." So two agents reading the same file are guaranteed to *see* the same text and guaranteed nothing about acting on it alike.

Workspace Rules and Activation Modes

Instruction files and rules can be scoped globally or to a specific workspace. Google Antigravity and Cursor extend basic instruction files by supporting explicit rule activation modes and workspace rules (`.agents/rules/` or `.cursor/rules/*.mdc`):

- **Always On**: Included unconditionally in every prompt context for the workspace (e.g. `GEMINI.md`, `AGENTS.md`, or `.mdc` files with `alwaysApply: true`).
- **Glob Scoped**: Activated automatically only when matching specific file patterns or paths in the workspace (e.g. `globs: ["src/ui/**/*"]` or Copilot's `applyTo`).
- **Model Decision**: Dynamically injected into context when the model judges the rule relevant to the current task or user prompt.
- **Manual**: Explicitly invoked by name or `@mention` during a session.

In Antigravity, workspace rules live under `.agents/rules/` (project-level) with global rules in `~/.gemini/GEMINI.md`, and workspace discovery operates via directory structures (`.agents/skills/` and `.agents/plugins/`).

Skills, and the Commands That Became Them

The distinction most people still draw here is out of date. Claude Code's documentation states:

> **Custom commands have been merged into skills.** A file at `.claude/commands/deploy.md` and a skill at `.claude/skills/deploy/SKILL.md` both create `/deploy` and work the same way.

Existing `.claude/commands/` files keep working, and if a command and a skill share a name, the skill wins. So "slash command versus skill" is now a question about **an older file layout versus a newer one**, not about two different capabilities.

What did *not* collapse is the invocation question. It moved into front matter, where it is now set per skill rather than implied by which directory the file sits in:

Front matter	You can invoke	The model can invoke
(default)	yes	yes
`disable-model-invocation: true`	yes	no
`user-invocable: false`	no	yes

This is the setting to think hardest about. A skill the model cannot invoke will never fire unless someone remembers it exists; a skill the model *can* invoke costs context on every turn, because its description sits in the listing whether or not it is ever used. Note also that `user-invocable` controls menu visibility rather than access: to block programmatic invocation, use `disable-model-invocation`.

The portable core is small and worth knowing exactly. The Agent Skills specification requires precisely two front matter fields — `name` and `description` — and says of the body that there are no format restrictions. Everything past that is a vendor extension: Claude Code's own docs describe invocation control, subagent execution, and dynamic context injection as extensions to the standard. Portability is therefore real but shallow: the folder and its metadata travel, and how much of the *behavior* travels depends on how alike two agents happen to be.

One concrete sign that the format genuinely crosses vendors: GitHub Copilot looks for skills in `.github/skills`, `.agents/skills`, and `.claude/skills` — a competitor's directory name.

Subagents: Delegating to a Fresh Context

Agent implementation and harness relationship cover what a subagent *is* and how the harness runs one. The authoring question is narrower: a subagent is a single Markdown file with front matter in `.claude/agents/` (project) or `~/.claude/agents/` (personal), whose body becomes that agent's entire system prompt.

Two details are easy to get wrong.

Precedence runs the opposite way from skills. For subagents, a project definition outranks a personal one. For skills, personal outranks project. If you keep a personal copy of something the repository also defines, which one wins depends on which mechanism you chose.

An @-mention picks the worker, not the words. Naming a subagent guarantees which one runs. It does not hand that subagent your sentence:

> Your full message still goes to Claude, which writes the subagent's task prompt based on what you asked. The @-mention controls which subagent Claude invokes, not what prompt it receives.

Hooks: The Part the Model Cannot Talk Its Way Around

Hooks are the mechanism this manual has not previously covered, and the one that changes what a customization is *worth*. They are defined in JSON settings files rather than in Markdown, and they run as shell commands, HTTP calls, or LLM prompts at fixed points in the harness's lifecycle. The page on instruction files draws the contrast plainly:

> Hooks execute as shell commands at fixed lifecycle events and apply regardless of what Claude decides to do.

An event fires, a matcher selects which handlers apply (by tool name, for instance), and the harness passes the handler JSON describing the event. Documented events include `SessionStart`, `UserPromptSubmit`, `PreToolUse`, `PostToolUse`, `Stop`, and `SessionEnd`, among a longer list; consult the reference rather than this page for the current set.

The practical rule: anything stated as "always" or "never" in a `CLAUDE.md` is a candidate to become a hook. Prose asks; a `PreToolUse` hook decides.

Settings and Permissions: The Boundary Around All of It

Settings live in `~/.claude/settings.json` (user), `.claude/settings.json` (project, shared), and `.claude/settings.local.json` (project, private), with a managed-policy layer above them for organizations. The precedence runs managed, then command-line arguments, then local, then project, then user.

One exception deserves emphasis, because it is the opposite of what the ladder implies:

> Permission rules behave differently because they merge across scopes rather than override.

Rules are written as `Tool(specifier)` — for example `Bash(npm run test *)`, `Read(./.env)`, or `Skill(commit)` — and sorted into `allow`, `ask`, and `deny`. Because a project's rules merge with yours rather than replacing them, a repository can tighten what you allow, and cannot quietly loosen it.

MCP Servers Are a Different Axis Entirely

Everything above changes what the agent *knows or must do*. An MCP server changes what it *can reach*: typed tools, data resources, and reusable templates exposed over a standard protocol. The specification is explicit that it "does not dictate how AI applications use LLMs or manage the provided context."

So MCP is never the answer to "how do I make the agent follow our convention", and always a candidate answer to "how do I let the agent query our issue tracker". Section 17 covers configuration and its failure modes.

Choosing

If you want to…	Use	Fires when
State a convention that should color everything	`CLAUDE.md` / `AGENTS.md`	every session, as context
Package a procedure the model should notice on its own	a skill	the model judges it relevant
Package a procedure *you* will invoke by name	a skill with `disable-model-invocation`	you type `/name`
Hand off self-contained work to a fresh context	a subagent	the model delegates, or you @-mention
Enforce something regardless of the model	a hook	a lifecycle event
Constrain what may run at all	permissions in settings	every tool call
Give the agent access to an external system	an MCP server	the model calls the tool

What Travels Between Tools

Mechanism	Portable?	Evidence
Agent Skills (`SKILL.md`)	yes, in format	open standard with a published specification, adopted across Claude Code, Codex, Copilot, Cursor, and Google Antigravity (and legacy Gemini CLI; see the harness landscape on Gemini CLI's sunset and folding into Antigravity CLI)
`AGENTS.md`	yes, as an open specification	standardizes filename, location, and precedence across Codex, Google Antigravity (and legacy Gemini CLI; see the harness landscape), Cursor, Aider, and Copilot (Claude Code reads `CLAUDE.md` by default, or imports `@AGENTS.md`)

Mechanism	Portable?	Evidence
MCP servers	yes	open protocol with multiple independent clients
`CLAUDE.md` / `GEMINI.md`	by courtesy	vendor instruction files read by default in their respective environments (Claude Code and Google Antigravity; see the harness landscape on the legacy Gemini CLI product folding into Antigravity CLI), and supported by courtesy in GitHub Copilot
Cursor Rules (`.cursor/rules/*.mdc`)	no	Markdown Cursor (`.mdc`) files with `alwaysApply` and `globs` frontmatter, scoped to Cursor
`.github/copilot-instructions.md`	no	GitHub Copilot only
`.github/instructions/*.instructions.md`	no	GitHub Copilot only, and not on every Copilot surface
`*.prompt.md` prompt files	no	Copilot only, and "only available in VS Code, Visual Studio, and JetBrains IDEs"
Hooks, settings, permissions	no	each harness defines its own

The lesson for a lab is to keep the portable layer carrying the meaning. Conventions belong in `AGENTS.md` and skills, which survive a change of tool; hooks and permissions are worth writing, and are worth writing as enforcement of rules that are also stated somewhere portable.

A Worked Example

The Morrison-Lab/ai-config repository is one lab member's configuration, versioned and synced across machines. Counting its `main` branch on 4 August 2026 — `git ls-tree -d --name-only origin/main skills/ | wc -l`, and the equivalent for the other directories — it carries:

- 177 skill directories
- 1 file under `commands/`, from before the merge described above
- 21 hook scripts, registered across `UserPromptSubmit`, `PreToolUse`, and `Stop`
- 7 subagent definitions

That distribution is itself the argument. Nearly everything is a skill, because a skill is the mechanism the model can reach for unprompted. The hooks are few and specific, because each one exists to make a rule that was being forgotten impossible to forget.

This repository is a smaller example of the same idea: it carries a `.github/copilot-instructions.md` for conventions that apply everywhere, plus path-scoped files under `.github/instructions/` whose `applyTo` globs attach them only when you edit a matching file.

8 How the Config Reaches a Machine

Section 7 describes *which* mechanism a customization should use. This section is about the step after that decision: how a config like a shared instruction repository actually reaches a machine, and how a broken install fails.

Two agents can load the identical instruction corpus and still behave differently, because behavior depends not only on what the config says but on how it is installed where the agent runs. An install problem is quiet by construction — nothing errors, the work still gets done, and a capability simply goes missing with no message that it existed.

Two Ways the Same Config Reaches a Machine

A repository of skills, hooks, and instruction files can be delivered two ways, and they are alternatives rather than layers.

- **As a plugin.** You enable the repository as a plugin from a marketplace, and the harness loads its skills, commands, and hooks directly from the plugin's own checkout at session start. It refreshes when the marketplace updates.
- **As a per-machine install.** You symlink (or copy) the repository's children into `~/.claude/` — `skills/`, `shared/`, `memories/`, `hooks/`, and `CLAUDE.md` — and register the hooks into `~/.claude/settings.json` by hand or with a script.

The trap is running both at once. The plugin path and the settings-file path carry different command strings, so the harness keeps both, and every hook fires twice. This fails in the safe direction — a doubled guard warns or blocks twice, it never goes missing — which is exactly why it is easy to leave in place unnoticed. Pick one path.

The `~/.claude` Layout

On a symlink-capable system, the children of `~/.claude` are symlinks into a working checkout of the repository, so a `git pull` in that checkout refreshes every skill and rule for free. Windows Git Bash is the common exception: without symlink privileges configured (Developer Mode,

or `MSYS=winsymlinks:nativestrict`), its `ln -s` falls back to real copies, so a pull does not propagate and the copies must be re-synced.

Two health checks answer two different questions, and a clean answer to one says nothing about the other.

- **Files.** Does `~/.claude/hooks/<script>` still track the checkout, or has it drifted, gone missing, or become a dangling link?
- **Bindings.** Does `settings.json` actually invoke that script on an event?

A guard can be a perfectly linked file that is registered to nothing, and an unregistered guard and a guard with nothing to block produce the same output: none. So verify both, and read the two results separately.

One Plugin Per Capability

Enabling the *same* repository as a plugin from two marketplaces — a personal fork and a lab org, say — loads its whole skill set into every session twice. That is not just untidy. A large instruction corpus is already a substantial share of a session's context, and a duplicated one can push a session far enough over the model's context limit to break subagent delegation, because a subagent independently re-loads that same duplicated skill set at start, and that alone can exceed the limit before any work begins. Keep one copy enabled.

Failure Modes and Their Symptoms

A broken install rarely announces itself; you read it backward from a symptom.

- **A stale hook reference blocks the tool it guards.** A `settings.json` entry pointing at a hook script that no longer exists errors before the guarded tool runs, so every call to that tool fails with the hook's error rather than the tool's output. A `PreToolUse` hook on `Bash` that references a missing script, for instance, blocks *all* shell commands until the entry is removed.
- **A dangling symlink silently drops a file.** A `~/.claude` child symlinked into a temporary worktree or a since-deleted clone becomes a dead link when that directory is removed, and the instructions or agent definition it provided quietly stop loading.
- **A check run against the wrong checkout cries wolf.** A verification that compares the installed copies against a stale or unintended checkout can report every entry as misdirected. That is a false alarm from a check aimed at the wrong target, not a fleet of real problems; re-run it against the checkout the install is actually anchored to before acting.

The common thread is that the install layer is a real surface, distinct from the content of the config, with its own failure modes and its own checks. When an agent behaves as though a rule or skill you wrote does not exist, suspect the install before you suspect the rule.

9 Agent Skills

Agent Skills are a lightweight, open standard for extending AI agent capabilities with specialized knowledge and workflows. The specification defines a portable, tool-agnostic format that any compatible agent can load.

What is a Skill?

At its core, a skill is a folder containing a `SKILL.md` file. This file includes the `name` and `description` metadata required by the specification, along with instructions that tell an agent how to perform a specific task. Skills can also bundle supporting resources:

- Scripts
- Reference documentation
- Templates and assets

```
my-skill/
   SKILL.md          # Required: metadata + instructions
   scripts/          # Optional: executable code
   references/       # Optional: documentation
   assets/           # Optional: templates, resources
```

How Agents Load Skills

The specification describes a *progressive disclosure* model, in which an agent typically loads a skill in three stages:

1. **Discovery**: At startup, the agent loads only the name and description of each available skill — just enough to know when it might be relevant.
2. **Activation**: When a task matches a skill's description, the agent reads the full `SKILL.md` instructions into context.
3. **Execution**: The agent follows the instructions, optionally executing bundled scripts or loading referenced files as needed.

This means full instructions load only when needed, so agents can maintain many skills with a small context footprint.

Why Use Skills?

Skills package procedural knowledge and team-specific context into portable, version-controlled folders. This gives agents:

- **Domain expertise**: Capture specialized knowledge as reusable instructions
- **Repeatable workflows**: Turn multi-step tasks into consistent, auditable procedures
- **Cross-tool reuse**: Build a skill once and use it across any skills-compatible agent

Further Reading

For the complete specification and more details, see agentskills.io.

A skill is one of several ways to customize an agent, and not always the right one. Section 7 compares it against:

- instruction files
- subagents
- hooks
- permissions

That section also explains why Claude Code's custom slash commands are now skills themselves.

The Morrison-Lab/ai-config repository contains an example of personal Claude Code configuration, including user-level skills, hooks, and subagents, synced across machines via Git.

10 Tessl: Skills Registry and CLI

Tessl is a commercial platform for managing Agent Skills (see Section 9 for what a skill is) the way `npm` or `pip` manage code dependencies: a searchable registry of skills and plugins, a `tessl` command-line tool that installs them into a project, and a hosted service that scores each skill for quality and security and measures whether it changes what an agent produces (Tessl 2026k). The notes below reflect the Tessl website and documentation as read on 2026-09-09.

What it is

Tessl's home page describes it as an "Agent Enablement Platform" and pitches it at enterprise teams whose developers have accumulated many skills with no inventory, no version control, and no security review (Tessl 2026i). The documentation lists six components (Tessl 2026k):

- a **registry and package manager** for discovering, installing, versioning, and rolling back skills and plugins

- **governance**: security scanning powered by Snyk, role-based access control, install and publish policies, and an audit trail
- **evals**, which run an agent on a task with and without a skill and compare the results
- **observability** of where skills activate across agent sessions
- an **inventory** that scans a GitHub organization and maps every `SKILL.md` across every repository
- the **Tessl Agent**, a conversational agent for driving the platform (in open beta)

The registry page counts more than 3,000 searchable skills, including Anthropic's own `docx` and `frontend-design` skills and skills published by OpenAI, Google, GitHub, and HashiCorp, each shown with a quality score, an "agent success vs baseline" multiplier from the evals, and a security-scan result (measured 2026-09-09) (Tessl 2026h).

The pages fetched for this review do not describe spec-driven development: apart from one blog article linked from the home page, the current site is about managing skills, not about generating code from specifications.

Relation to the Agent Skills standard

Tessl does not define its own skill format. A Tessl skill is the same `SKILL.md` folder the Agent Skills specification defines, and Tessl's conformance review checks a published skill against that specification (Tessl 2026f). What Tessl adds is a lifecycle around the standard: a manifest (`tessl.json`) that records which skills a project depends on, versioned publishing with `--bump patch|minor|major`, and `tessl outdated` and `tessl update` for keeping dependencies current (Tessl 2026j).

Tessl distinguishes two kinds of context (Tessl 2026a):

- **rules**, which apply to every task (the role instruction files play in Section 7)
- **skills**, which load only when the task matches their description

A **plugin** bundles skills and rules together, and can also carry MCP servers and hooks, so it is roughly the same unit as a Claude Code plugin.

The registry and the Posit skill collection in Section 11 are two different distribution routes for the same kind of artifact. Posit publishes its skills as a Git repository that the `skills` CLI or Claude Code's plugin marketplace installs from; Tessl installs from a GitHub repository URL as well as from its own registry (the documentation's example is `tessl install https://github.com/anthropics/skills`), and either way writes the skill into the agent's skills directory.

Installing and using it

The CLI installs natively (`curl -fsSL https://get.tessl.io | sh`), through Homebrew, or on Windows through `winget install tessl.tessl`; the `npm` package is deprecated (Tessl 2026b). If your organization's policy requires it, download and inspect the script before running it rather than piping it straight to `sh`. It needs Node.js 22.17 or later (Tessl 2026e), and by default it checks for updates every three hours and installs them silently (Tessl 2026b). Logging in (`tessl login`) uses a GitHub or Google account and is optional for searching and installing.

In a project, `tessl init` detects the coding agents present and configures each of them. The supported list is Claude Code, Cursor, Codex, Gemini CLI, Antigravity, GitHub Copilot CLI, and GitHub Copilot for VS Code, with manual setup documented for other MCP-capable agents such as OpenCode and OpenClaw (Tessl 2026e). `tessl search "code review"` searches the registry by meaning rather than by keyword, and `tessl install <workspace>/<skill>` installs a skill into `.tessl/plugins/` and links it into each agent's directory, so for Claude Code the skill appears under `.claude/skills/` (Tessl 2026f). Adding `--global` installs into `~/.tessl/` for every project instead. On Windows the links are directory junctions, so moving the project folder requires running `tessl install` again.

Before installing a third-party skill, `tessl review run security ./path-to-skill` scans it and returns a Snyk severity from `LOW` to `CRITICAL`; `--fail-on high` makes the command exit non-zero for use in CI (Tessl 2026c). Installing from the registry warns on critical and high findings but leaves the decision to the user, unless a workspace administrator has set an install policy that blocks them.

Pricing

Tessl charges by usage credits rather than by seat. Publishing and installing skills and plugins is always free; credits pay for reviews, evals, and Tessl Agent sessions, and frontier models consume credits faster than the defaults. The tiers as of 2026-09-09 (Tessl 2026g):

- **Free**: 1,000 credits per month, one workspace, free best-practice reviews on publicly published plugins
- **Team**: $100 per month for 5,000 credits, top-ups, spending limits, and role management within a workspace
- **Enterprise**: custom pricing with multiple workspaces, SAML single sign-on, install and publish controls, mandatory skills, and bring-your-own-key models

The CLI collects telemetry by default, and the documentation says this may include conversations, code snippets, and file contents; it can be turned off in the CLI configuration (Tessl 2026d). Tessl also reserves the right to train on data from free-tier usage, though it states it has not done so (Tessl 2026d).

Useful to us? Marginally, for vetting skills; not as a package manager

The lab's skills already live in Morrison-Lab/ai-config, where they are version-controlled, reviewed in pull requests, and installed as a Claude Code plugin, and the Posit skills the lab uses are copied into that repository (see Section 11). Tessl's package-manager features solve a problem the lab does not have: one repository is the single source of skills, so there is no sprawl to inventory, and the governance, role, and policy features are built for organizations with many teams rather than one. Adopting the CLI would add a Node.js dependency, a `tessl.json` manifest and `.tessl/` directory to each project, and telemetry that includes code by default.

Two parts are worth using without adopting the rest. The registry is a convenient place to browse public skills and see a security-scan result and an eval score before copying one, and the free `tessl review run security` scan is a quick check on a skill from an unfamiliar source before it goes into `ai-config`. The eval model, running the same task with and without a skill and comparing the results, is the only part of Tessl the lab has no equivalent for, and it is worth keeping in mind if the lab ever wants evidence that a skill helps rather than an impression that it does.

11 The Posit Skill Collection

posit-dev/skills is the collection of Agent Skills published by Posit (see Section 9 for what a skill is) for R, Python, Quarto, and Shiny work. It is MIT-licensed, and its skills load in Claude Code, Claude.ai, or through the Claude API, or in any other skills-compatible agent. The notes below reflect the repository's README as read on 2026-09-01.

What it contains

The skills are grouped into eight categories, each of which can be installed on its own:

- **posit-dev**: general developer skills, including an adversarial `critical-code-reviewer`, `describe-design` for architecture documentation with Mermaid diagrams, and `review-testing` for auditing test code after a change
- **github**: pull-request workflows, such as `pr-create`, which opens a PR and then watches CI and debugs failures until it passes, and `pr-threads-address` / `pr-threads-resolve` for review threads
- **open-source**: `create-release-checklist` and `release-post` for R and Python package releases
- **r-lib**: `testing-r-packages` (`testthat` 3+), `cli`, `cran-extrachecks`, `lifecycle`, `r-package-development`, `mirai` (asynchronous and parallel R), and `alt-text`
- **ggsql**: writing queries in `ggsql`, a grammar of graphics for SQL
- **shiny**: `brand-yml`, `shiny-bslib`, and `shiny-bslib-theming`

- **quarto**: `brand-yml`, `authoring` (including migration from R Markdown and `bookdown`), and `alt-text`
- **connect**: `deploy-to-connect` for Posit Connect

Each skill carries an `author`, `version`, and `license` in its `SKILL.md` frontmatter, so a copied skill stays attributable.

How to install it

The README documents these routes:

- the cross-agent `skills` CLI (`npx skills add posit-dev/skills --list`, `--all`, or `--skill <name>`), which targets Claude Code, Codex, Cursor, Cline, and others
- in Claude Code, adding the repository as a plugin marketplace (`/plugin marketplace add posit-dev/skills`) and then choosing categories in the plugin UI
- in Claude Code, installing a category directly (`/plugin install <category>@posit-dev-skills`)
- copying skill folders into the agent's skills directory by hand
- uploading a skill to Claude.ai, or loading it through the Skills API

Once installed, the agent activates a skill from its description rather than on an explicit command, which is the progressive-disclosure model Section 9 describes.

Useful to us? Yes, and largely already adopted

The R and Quarto skills match this lab's daily work, and nine of them are already copied into Morrison-Lab/ai-config, adapted for that repository's conventions with the upstream MIT license text kept alongside them:

- `brand-yml`
- `cli`
- `cran-extrachecks`
- `create-release-checklist`
- `lifecycle`
- `quarto-authoring`
- `r-package-development`
- `release-post`
- `testing-r-packages`

Copying the skills in rather than installing the marketplace plugin lets them follow the lab's own review and hook conventions, at the cost of tracking upstream changes by hand. Four categories are not adopted. Three of them have the least overlap with lab work: Shiny app development, `ggsql`, and Posit Connect deployment. The fourth, `posit-dev`, overlaps with practices the lab already has: its `critical-code-reviewer` parallels the adversarial self-review subagent that

the lab's agent configuration dispatches before every push, and `new-work` / `working-on` keep a per-task tracking document where the lab uses the issue tracker and a session notebook. The `github` skills likewise overlap with the lab's own pull-request workflow and with the review setup in Claude Code Action review, so both categories are worth reading for ideas rather than installing alongside that tooling. The contribution guidance in the repository recommends Anthropic's `skill-creator` skill for authoring new skills.

12 Posit AI Tooling on GitHub

The `posit-dev` GitHub organization holds the open-source work of Posit PBC that does not live under the `rstudio`, `tidyverse`, `r-lib`, or `quarto-dev` organizations. Much of the LLM tooling from Posit started there, so this section reviews the organization for repositories relevant to working with AI. The Posit skill collection, `posit-dev/skills`, is covered separately in Section 11 and is not repeated here.

The listing below was derived from the GitHub API (`gh repo list posit-dev --limit 300`) rather than from browsing the website. The organization had 226 public repositories, of which the 16 below are about building with or evaluating LLMs (measured 2026-09-09). Star counts and last-push dates are as of the same date. The rest of the organization is Positron and its extensions, Shiny for Python, `great-tables`, `pointblank`, `air`, `ggsql`, Posit Team deployment tooling, container images, and workshop material.

AI-related repositories

Repository	Purpose	Language	Stars	Last push	Verdict for this lab
`querychat` (Posit PBC 2026m)	Natural-language exploration of tabular data, translated to SQL, for R and Python Shiny apps	Python, R	211	2026-09-08	Maybe: a safe pattern for letting an LLM query a data frame

Repository	Purpose	Language	Stars	Last push	Verdict for this lab
`mcptools` (Posit PBC 2026i)	Model Context Protocol for R: expose a running R session as an MCP server, or register MCP servers with `ellmer` chats	R	194	2026-08-22	Yes: the MCP layer `btw` uses, and the way to expose a lab function as a tool
`chatlas` (Posit PBC 2026e)	Python client for LLM chat and tool calling, the Python counterpart of `ellmer`	Python	175	2026-09-08	No: the lab builds in R
`shinychat` (Posit PBC 2026p)	Chat UI component for Shiny, in R and Python	TypeScript, R, Python	137	2026-09-09	Maybe: only if a lab app needs a chat panel
`btw` (Posit PBC 2026d)	Toolkit for giving LLMs context about an R session: copy context to the clipboard, chat in the IDE, or connect R to coding agents through `mcptools`	R	135	2026-09-09	Yes: the most direct fit for an R-centric workflow

Repository	Purpose	Language	Stars	Last push	Verdict for this lab
`mcp-repl` (Posit PBC 2026h)	Sandboxed MCP server that keeps a persistent R or Python session alive across an agent's tool calls, with plots returned as images	Rust	65	2026-07-27	Yes, untried: replaces `Rscript -e` round trips in Claude Code
`shiny-assistant` (Posit PBC 2026o)	Source of the hosted Shiny Assistant chat app	Python	58	2026-01-17	No: an application, not a library
`raghilda` (Posit PBC 2026n)	Retrieval-augmented generation pipeline in Python, with DuckDB or ChromaDB storage	Python	36	2026-09-02	No: the R counterpart is `tidyverse/ragnar`
`shinyrealtime` (Posit PBC 2026q)	Voice chat in Shiny apps via the OpenAI Realtime API	Python, R, TypeScript	25	2026-07-31	No
`commons` (Posit PBC 2026g)	Experimental framework for data agents built on trusted R or Python calculations and data dictionaries	R, Python	20	2026-09-09	Watch: experimental, but aimed at the "agent over our own analyses" problem

Repository	Purpose	Language	Stars	Last push	Verdict for this lab
code-index (Posit PBC 2026f)	Semantic code search over a repository, exposed to coding agents through MCP; parses R, Python, Quarto, and Markdown	Go	6	2026-04-29	Maybe: for large codebases only
ai-lib (Posit PBC 2026a)	TypeScript LLM-provider infrastructure behind Posit Assistant, successor to the archived ai-provider-bridge	TypeScript	5	2026-09-08	No: internal infrastructure
bluffbench2 (Posit PBC 2026c)	Evaluation of whether agents notice subtle data-quality artifacts in plots, implemented with vitals	R	4	2026-07-21	Yes, as a reference: an eval built in R about data-analysis judgment
positron-copilot-chat (Posit PBC 2026l)	Fork of Microsoft's Copilot Chat extension bundled into Positron	TypeScript	4	2026-05-06	No: consumed through Positron itself

Repository	Purpose	Language	Stars	Last push	Verdict for this lab
assistant-feedback (Posit PBC 2026b)	Issue tracker for feedback on Posit Assistant	none	4	2026-05-28	Reference: where to report Positron Assistant problems
nesevals (Posit PBC 2026j)	Evaluation tooling for next-edit-suggestion prompts and models	R	2	2026-04-07	No: relevant to IDE builders

Two tools the lab might expect to find here live elsewhere. ellmer, the R package for LLM chat and tool calling that btw and querychat build on and that mcptools and shinychat integrate with, and ragnar, its retrieval companion, are in the tidyverse organization. Positron Assistant has no public repository of its own. Going by the repository descriptions, its provider layer is ai-lib, the Copilot fork above supplies its chat surface, and the rest is part of the positron repository (Posit PBC 2026k). The organization also holds workshop and demo repositories (positron-ai-workshop, ai-powered-app-workshop, shiny-querychat-workshop, posit-conf-chat) that are worked examples of the packages above rather than tools.

Useful to us? Yes, three tools and one reference

The lab's workflow is R and Quarto driven by Claude Code and Copilot, so the question is which of these repositories connect that workflow to an R session.

- **btw** is the direct fit. Its clipboard mode pastes the structure of the objects in an R session into whatever chat is open, and its MCP server route (through mcptools) lets Claude Code read package documentation and inspect the global environment instead of guessing at column names. The MCP route follows the setup pattern in Section 17.
- **mcptools** is the layer underneath btw, and the package to use directly when a lab package wants to expose one of its own functions to an agent as a tool.
- **mcp-repl** addresses a cost the lab already pays: each Rscript -e call an agent makes starts a fresh process, so data and packages are reloaded on every check. A persistent, sandboxed session with inline plots is a better fit for iterating on an analysis with an agent, at the cost of a Rust binary to install. It has not yet been tried in the lab.

- **`bluffbench2`** is a reference rather than a dependency. It is an evaluation written in R with `vitals` that measures whether an agent notices data-quality problems, which is the kind of judgment the lab expects a human reviewer to keep. Anyone building an eval of the lab's own agent configuration can start from its harness.

`querychat` and `shinychat` matter only if the lab ships Shiny apps with a chat interface. `commons` is a framework for data agents rather than a chat component, and it is still marked experimental. The Python packages (`chatlas`, `raghilda`) duplicate R tools the lab already prefers. Positron Assistant is configured through Positron rather than through these repositories; see Section 1 for its OpenAI setup and running agents offline for running it against Ollama.

13 Useful plugins

This site's Quarto sources already use Semantic Line Breaks (SemBr): a line break after each substantial unit of thought, so the source is easier to edit while the rendered HTML still reads as ordinary paragraphs.

sembr/skills packages that convention as Agent Skills for any skills-compatible tool. Install it one of these ways:

- Claude Code: `/plugin marketplace add sembr/skills`
- Cursor: from the Cursor Marketplace, or **Settings > Rules > Add Rule > Remote Rule (Github)** with `sembr/skills`
- skills CLI: `npx skills add https://sembr.org`
- Pi: `pi install git:github.com/sembr/skills`

Ask the agent to apply SemBr on new or revised prose (the `sembr-reformat` skill); there is no need to reformat an entire document in one pass.

ponytail makes the agent think like the laziest senior dev in the room: the best code is the one you never write. Before writing, the agent walks a seven-rung ladder:

1. does this need to exist (YAGNI);
2. is it already in the codebase;
3. is it in the stdlib;
4. is it a native platform feature;
5. is it an installed dependency;
6. can it be one line;
7. only then write the minimum that works.

The agent reads the touched code first and is lazy about the solution, never about reading, and never cuts validation, error handling, security, or accessibility. Measured on twelve real Claude Code tasks in a FastAPI + React repo (Haiku 4.5, n=4), it averages about 54 percent less code (up to 94 percent where the agent would otherwise overbuild, for example a date picker)

with about 20 percent lower cost and 27 percent faster, while staying fully safe. It installs as a plugin or skill for more than twenty agents (Claude Code, Codex, Copilot, Cursor, OpenCode, Gemini, and others) and works from a checkout via `AGENTS.md` where a plugin is not needed.

Contextify keeps your Claude Code and Codex history forever in a private, searchable timeline. Claude Code deletes history after 30 days; Contextify watches both tools, summarizes each message (on-device via Apple Intelligence on macOS 26, or Lite Mode on macOS 15), and lets you search every conversation you ever had. It runs local-first with no account required, and optionally syncs across devices via Cloud Sync or a self-hosted instance you operate. The ambient timeline lets you follow sessions in real time or skim what happened while you were away.

The lab's portable agent config lives in `Morrison-Lab/ai-config`. It is a plugin-or-symlink install of skills, hooks, and memories — not a third marketplace next to SemBr. How that config actually reaches a machine, and how a doubled plugin install fails, is Section 8. Section 7 is the worked example of what the corpus contains. Section 14 maps the official and community marketplaces these four sit in and sets each against `ai-config`.

14 The plugin catalog

Section 13 names four plugins the lab already uses. This section is the map they sit on: what Anthropic's official marketplace contains, which community plugins are worth knowing about, and how each of them relates to the lab's own `Morrison-Lab/ai-config` (Morrison Lab 2026a). It closes with the question that prompted it: what the `ralph-loop` plugin is, and whether it is the same thing as `ai-config`'s `ardi`.

Plugin counts and names change weekly, so every figure below is stamped with the date it was read. The anatomy of a plugin bundle (skills, hooks, MCP servers, agents, commands) is Section 16; this section assumes it.

The three Anthropic marketplaces

Anthropic runs three plugin catalogs for Claude Code, and they differ in who curates them (Anthropic 2026d):

- **`claude-plugins-official`** (`anthropics/claude-plugins-official` (Anthropic 2026b)). Curated by Anthropic at its discretion; there is no application process. Claude Code registers it automatically the first time it starts interactively, so `/plugin install <name>@claude-plugins-official` works with no setup. Browse it with `/plugin` (the **Discover** tab) or at claude.com/plugins.

- `claude-community` (anthropics/claude-plugins-community (Anthropic 2026c)). Third-party submissions that passed Anthropic's automated validation and safety screening, each pinned to a commit SHA and synced nightly from the review pipeline. Added by hand: `/plugin marketplace add anthropics/claude-plugins-community`, then `/plugin install <name>@claude-community`. Pull requests against the mirror are closed automatically; submissions go through a form.
- `claude-code-plugins` (anthropics/claude-code, plugins/ (Anthropic 2026a)). The demo marketplace: thirteen example plugins (`commit-commands`, `code-review`, `feature-dev`, `hookify`, `ralph-wiggum`, and others) that show what the plugin system can do. Most of them also ship in the official marketplace under the same or a renamed entry (`ralph-wiggum` there is `ralph-loop` here).

The official README carries the warning that governs all three: Anthropic does not control what MCP servers, files, or other software a plugin includes, and cannot verify that it works as intended or will not change (Anthropic 2026b). A plugin runs arbitrary code with your user privileges, so the trust decision is yours, per plugin, every time.

What the official marketplace contains

The official catalog's `marketplace.json` lists 287 plugins (measured 2026-09-09 from the copy Claude Code caches locally; (Anthropic 2026b)). Only 38 of them live in the repository's own `plugins/` directory, which the README describes as the plugins Anthropic develops and maintains. Another 15 sit in `external_plugins/`, and the remaining 234 are pointers to third-party repositories (Amazon, Microsoft, Google's `gemini-cli-extensions`, Databricks, Hugging Face, Sentry, and many vendors), each pinned to a commit SHA. So "official" mostly means "listed and pinned by Anthropic", not "written by Anthropic". By declared category the catalog is:

- development (119)
- productivity (49)
- database (38)
- monitoring (20)
- security (18)
- a long tail of deployment, design, automation, learning, location, testing, migration, and math

The 38 Anthropic-maintained plugins fall into five groups. The verdicts are for this lab's work (R packages, Quarto sites, Python and shell tooling, GitHub pull requests), not for software teams in general.

Code intelligence

Twelve `*-lsp` plugins (`clangd`, `csharp`, `gopls`, `jdtls`, `kotlin`, `lua`, `php`, `pyright`, `ruby`, `rust-analyzer`, `swift`, `typescript`) connect a Language Server Protocol server so that Claude sees type errors and missing imports after every edit and can jump to definitions instead of grepping (Anthropic 2026d). The plugin does not install the language server binary; you do, and cloud sessions never start it.

Verdict: `pyright-lsp` is worth installing wherever the lab writes Python. There is no R entry in the catalog (measured 2026-09-09), so R work gets no diagnostics from this route; the docs describe an `.lsp.json` for writing your own LSP plugin, which is the path if anyone wants to wire up R's `languageserver`.

Development workflows

- `commit-commands`: commit, push, and open a PR from slash commands.
- `code-review`: multi-agent PR review with confidence-based scoring.
- `pr-review-toolkit`: reviewer agents specialized by concern (comments, tests, error handling, type design).
- `feature-dev`: a phased explore-design-implement workflow with dedicated agents.
- `code-simplifier`: an agent that refines code for clarity without changing behaviour.
- `code-modernization`: a preflight-assess-transform workflow for legacy codebases.
- `security-guidance`: pattern warnings on every edit plus an LLM diff review at `Stop`, with fixes applied in the same session.
- `claude-security`: deeper vulnerability scanning of your own code, at a chosen effort level.
- `ralph-loop`: the self-referential iteration loop, treated on its own below.

Verdict: `security-guidance` fills a gap `ai-config` does not cover and costs little, so install it. `commit-commands`, `code-review`, and `pr-review-toolkit` overlap the lab's forge workflow (claim, PR-on-claim, `ardi`, the `@claude` review action) and would run beside it rather than replace it; skip them unless you are working outside a lab repository. `feature-dev` and `code-modernization` are aimed at application codebases and have not earned a place here.

Building your own extensions

- `plugin-dev`: seven skills covering hooks, MCP integration, and plugin structure.
- `skill-creator`: create, improve, and benchmark skills.
- `hookify`: write a hook from a conversation pattern or an explicit instruction.
- `mcp-server-dev` and `agent-sdk-dev`: building MCP servers and Claude Agent SDK programs.
- `mcp-tunnels`: reach a private MCP server through an Anthropic tunnel.
- `claude-md-management`: audit `CLAUDE.md`, capture session learnings, keep project memory tidy.

- `claude-code-setup`: analyze a codebase and recommend hooks, skills, and MCP servers for it.

Verdict: `skill-creator` and `plugin-dev` are the ones to reach for when adding to `ai-config`, since the corpus is itself a plugin. `claude-md-management` duplicates `ai-config`'s `ums` and `memorize` skills and its `MEMORY.md` conventions; skip it in lab repositories. `hookify` generates a hook from a description; `ai-config`'s hooks are hand-written Python with a test per hook, so use `hookify` for a personal one-off and the corpus for anything shared.

Output styles and reports

`explanatory-output-style` recreates the deprecated built-in Explanatory style (commentary on implementation choices), and `learning-output-style` implements a Learning style that never shipped (prompts for you to write key pieces yourself). `receipts`, `session-report`, and `project-artifact` generate HTML reports of what you shipped, what a session cost, and a project status page. `playground` builds single-file interactive HTML explorers; `frontend-design` targets distinctive web interfaces; `math-olympiad` solves competition mathematics with adversarial verification; `cwc-makers` sets up a hardware kit.

Verdict: `learning-output-style` is a reasonable choice for a student who wants to learn a codebase rather than delegate it, per when to use AI. `session-report` is useful when a quota question comes up. The rest are not relevant to lab work.

External integrations

The `external_plugins/` directory and many third-party entries bundle a pre-configured MCP server: `github`, `gitlab`, `atlassian`, `asana`, `linear`, `notion`, `figma`, `slack`, `sentry`, `vercel`, `firebase`, `supabase`, `context7`, `playwright`, `serena`, `terraform`, and messaging bridges for Discord, Telegram, and iMessage (Anthropic 2026d).

Verdict: `github` is the one that matters here; Section 17 covers configuring it, and PR activity notifications what it adds. `context7` (live library documentation lookup) is worth a try for Python and JavaScript work. Everything else depends on whether the lab uses the service.

Community plugins worth knowing

Beyond the three Anthropic catalogs, any GitHub repository with a `.claude-plugin/marketplace.json` is a marketplace (`/plugin marketplace add owner/repo`), and the four plugins in Section 13 (`sembr/skills`, `ponytail`, Contextify, and `ai-config` itself) all reach you that way. One more is widely enough used to describe here.

Superpowers (`obra/superpowers` (Vincent 2026)) calls itself "a complete software development methodology for your coding agents, built on top of a set of composable skills". The 6.3.0 build installed on one lab machine carries fourteen skills (measured 2026-09-09):

- `brainstorming`
- `writing-plans` and `executing-plans`
- `test-driven-development`
- `systematic-debugging`
- `subagent-driven-development` and `dispatching-parallel-agents`
- `requesting-code-review` and `receiving-code-review`
- `using-git-worktrees`
- `finishing-a-development-branch`
- `verification-before-completion`
- `using-superpowers` and `writing-skills`, the two meta-skills

The `using-superpowers` skill loads at session start so the others trigger on their own. It is MIT-licensed and installs into more than a dozen agents, among them:

- Claude Code, via `/plugin install superpowers@claude-plugins-official`
- Cursor, Codex, and Copilot CLI
- Gemini CLI, OpenCode, and Antigravity
- Pi and Hermes

Verdict: install it if you want an opinionated end-to-end process and are not already running `ai-config`. Running both means two session-start bootstraps competing to set the workflow, and `ai-config` already covers the same ground in its own vocabulary (`brainstorm`, `st`, `ardi`, `adversarial-reviewer`, `clean-worktrees`, `wrap-up`). The skills that do not overlap (`systematic-debugging`, `test-driven-development`) are worth reading even if you do not install the plugin.

The community marketplace itself is small so far (four plugin directories visible on 2026-09-09: `eli5`, `quickdesign`, `testdino`, `tres-finance-plugin`; (Anthropic 2026c)), so "branching out" today means the official catalog's third-party entries and independent repositories, not that mirror.

Compared with `ai-config`

`Morrison-Lab/ai-config` (Morrison Lab 2026a) is the lab's own plugin. The copy Claude Code caches on one lab machine holds 198 skills and 90 hook scripts (measured 2026-09-09), plus memories and the `shared/` fragments this site vendors into fully clean and its neighbours. It installs as a Claude Code plugin (`/plugin marketplace add Morrison-Lab/ai-config`), as a Cursor plugin, and via `bootstrap.sh` for Codex, Gemini CLI and Antigravity, VS Code Copilot, and OpenCode (Section 8).

Set beside the marketplaces above, four differences decide what to install:

- **Scope.** A marketplace plugin does one thing: one language server, one MCP server, one workflow. `ai-config` is a whole working style: forge etiquette (claim, PR-on-claim, `ardi`, `mwc`), lab coding and writing conventions, quota management, and the memory

that carries lessons between sessions. Nothing in the official catalog is shaped like that; `superpowers` is the closest, and it stops at the software-methodology layer.
- **Enforcement.** The official workflow plugins are mostly skills and agents, which advise. `ai-config` pairs its skills with hooks that block: an unauthorized merge, a force push, a reply that promises without a mechanism. The one official plugin with a comparable `Stop`-hook design is `security-guidance`, which is why it is the clearest addition.
- **Cross-agent reach.** Both `ai-config` and `superpowers` install into several agents from one repository; the official plugins are Claude Code only. If a lab member works in Codex or Antigravity as well, that decides it.
- **Ownership.** A finding in an official plugin is a bug report to Anthropic; a finding in `ai-config` is a PR you can open today, and the corpus's UMS discipline expects exactly that.

The practical composition, as of 2026-09-09: `ai-config` as the base, `pyright-lsp` and `security-guidance` from the official marketplace, `github` where the MCP server is not already configured by hand, and `sembr/skills` for prose repositories. Everything else is a per-person choice.

What is `ralph-loop`, and is it `ardi`?

`ralph-loop` (official marketplace; `ralph-wiggum` in the demo marketplace) packages Geoffrey Huntley's "Ralph" technique (Anthropic 2026e; Huntley 2025). Ralph, in Huntley's words, "is a Bash loop":

```
while :; do cat PROMPT.md | claude-code ; done
```

The same prompt file is fed to a fresh agent run, over and over. Nothing changes between iterations except the repository: the previous run's files, commits, and test results are what the next run reads. Tests supply the backpressure that keeps each pass honest, and the operator "tunes Ralph by adding a sign" to the prompt when a failure pattern shows up (Huntley 2025). Huntley says it works best on greenfield projects, one task per loop (Huntley 2025).

The plugin moves that loop inside a single Claude Code session. `/ralph-loop "<prompt>" --max-iterations <n> --completion-promise "<text>"` installs a `Stop` hook that intercepts the agent's attempt to end the turn and feeds the same prompt back, until the agent's output contains the completion-promise string exactly, or the iteration cap is hit, or you run `/cancel-ralph` (Anthropic 2026e). The caveats in the README are the important part: the promise is an exact-string match, so it cannot distinguish "done" from "blocked", and `--max-iterations` (unlimited by default) is the only real safety net. It lists tasks needing human judgment, one-shot operations, unclear success criteria, and production debugging as cases where not to use it.

So: is it like `ardi`? Both are loops that refuse to let the agent stop early, and both rely on a `Stop`-time mechanism (`ai-config`'s own `Stop` hooks are what block a placeholder reply or an empty promise). Past that they answer different questions.

	`ralph-loop`	`ardi`
What it loops over	one prompt, re-fed to the same session	one pull request, across review rounds and sessions
Who decides "done"	the agent, by emitting an agreed string	an independent reviewer's verdict plus green CI on the current head
What changes each round	nothing but the repository state	the reviewer's new findings, each Addressed, Rebutted, or Deferred
Safety valve	an iteration cap you set	a human merge gate; the loop reports ready and never merges
Where it fits	before a PR exists: grind an implementation against tests	after a PR exists: drive it to fully clean

`ardi` is not self-referential. Each iteration is triggered by something outside the agent (a review landing, a check turning red), it re-arms itself with a timer rather than a `Stop` hook, and its terminal condition is fully clean, which an agent cannot declare about itself. `ralph-loop` has no reviewer in the loop at all; its judge is whichever tests the prompt tells it to run.

The two compose rather than compete. A Ralph loop is a reasonable way to get a well-specified, well-tested change to the point of opening a PR; `ardi` takes over from there. What the lab should not do is use `ralph-loop` as a substitute for review: a loop whose exit is a string the agent writes will exit whether or not the work is right.

15 Running Codex Inside Claude Code: the codex-plugin-cc Plugin

`openai/codex-plugin-cc` is OpenAI's official Claude Code plugin for running Codex from inside a Claude Code session (OpenAI 2026a). It adds slash commands that ask Codex to review the current work or take over a task, while Claude Code stays the harness the user is typing into. The repository is Apache-2.0 licensed, has about 33,000 GitHub stars, and its latest release is `v1.0.6` from 2026-07-08 (measured 2026-09-09).

What it does

The plugin does not ship a second Codex runtime. It wraps the `codex` binary already installed on the machine, talking to it through the Codex app server, so it uses the same login, the

same `config.toml`, and the same repository checkout that the Codex CLI would use directly (OpenAI 2026a). A Node.js script (`codex-companion.mjs`) does the actual work; the Markdown command files tell Claude to invoke that script once and to return Codex's output verbatim rather than paraphrasing it. The command files enforce this: they forbid Claude from fixing anything a Codex review reports until the user says which findings to act on.

Installation

Node.js 18.18 or later is required (OpenAI 2026a). The plugin is installed through Claude Code's plugin marketplace mechanism (see Section 16 for how marketplaces and manifests fit together); the repository's `marketplace.json` names the marketplace `openai-codex`, which is why the install command does not repeat the repository name:

```
/plugin marketplace add openai/codex-plugin-cc
/plugin install codex@openai-codex
/reload-plugins
/codex:setup
```

`/codex:setup` checks whether Codex is installed and logged in, and offers to run `npm install -g @openai/codex` when it is missing. Logging in is done outside the plugin with `codex login`.

What gets added

The marketplace manifest lists a single plugin, `codex`, whose bundle contains commands, one subagent, three skills, and a hook file (OpenAI 2026a).

Slash commands:

- `/codex:review` runs Codex's built-in read-only review on the working tree or on the branch against `--base <ref>`, with `--scope` to force one or the other and `--wait` or `--background` to choose whether it blocks the session. It takes no focus text.
- `/codex:adversarial-review` is the steerable version: it questions the design and assumptions rather than only the diff, accepts free-text focus such as "look for race conditions", and takes the same `--base`, `--scope`, `--wait`, and `--background` flags.
- `/codex:rescue` hands a task to Codex to investigate or fix, with `--model`, `--effort`, `--resume`, and `--fresh` flags. By default this run is write-capable.
- `/codex:transfer` exports the current Claude Code transcript into a Codex thread and prints the `codex resume <session-id>` command, for continuing the same conversation in Codex.
- `/codex:status`, `/codex:result`, and `/codex:cancel` manage background jobs.
- `/codex:setup` checks the install and toggles the review gate described below.

Subagent:

- `codex:codex-rescue` (pinned to the Sonnet model tier) is a thin forwarder that `/codex:rescue` invokes. Its definition tells it to make exactly one `Bash` call to the companion script and to do no repository inspection or independent reasoning of its own, beyond using the `gpt-5-4-prompting` skill to tighten the forwarded prompt.

Skills (all marked `user-invocable: false`, so Claude loads them by description and the user cannot call them by name):

- `codex-cli-runtime`, the calling contract for the companion script, attached to the subagent
- `gpt-5-4-prompting`, guidance for tightening a request into a block-structured Codex prompt, attached to the subagent
- `codex-result-handling`, rules for presenting Codex output without altering it, attached to no agent and so available to the main session

Hooks:

- `SessionStart` and `SessionEnd` hooks record the transcript path that `/codex:transfer` later reads.
- An optional `Stop` hook implements a **review gate**: when enabled with `/codex:setup --enable-review-gate`, every time Claude tries to end a turn, Codex reviews that turn and blocks the stop if it finds problems. The README warns that this can produce a long Claude-Codex loop that drains usage limits quickly, and recommends enabling it only in an actively monitored session (OpenAI 2026a).

Authentication and cost

The plugin uses whatever `codex login` set up. Two routes exist (OpenAI 2026a, 2026b):

- A **ChatGPT account** (including the Free tier) draws on the Codex usage limits included in that plan; every review or rescue run counts against them.
- An **OpenAI API key** bills at API rates, which OpenAI positions for shared or automated environments such as CI.

Either way, the Claude Code session itself still bills to the Anthropic plan. A rescue run therefore spends on both providers at once: Claude's tokens to dispatch and read back, and Codex's usage limits or API credits to do the work. Model and reasoning-effort defaults come from `~/.codex/config.toml` or a project-level `.codex/config.toml`, the latter only in a trusted project (OpenAI 2026a).

Comparison with the lab's existing Codex paths

The lab already reaches Codex two ways, and the plugin overlaps with both.

`delegate-to-codex` in `ai-config` (Morrison Lab 2026b). That skill runs `codex exec` directly from a Bash call, with a read-only sandbox by default, prompts written to files, an optional JSON output schema, and a background runner that fans out several prompts at once and polls a completion marker. It exists to spend the separately billed ChatGPT plan on heavy read/draft/verify fan-out before Claude's own quota. The plugin covers the single-task case of that skill and adds three things the skill lacks: job tracking, threads that can be resumed, and a subagent Claude can call proactively. It does not cover the fan-out case, it defaults to write-capable runs where the skill defaults to read-only, and it does not enforce structured output. The two also disagree on who orchestrates: the skill keeps Claude as the integrator that assembles Codex's parts, while the plugin's result-handling skill tells Claude to relay Codex's answer and stop.

Codex as a GitHub reviewer (Codex GitHub review). That path runs through Codex Cloud and posts a review on the pull request, so it needs a connected repository and a workspace that permits Codex Cloud. `/codex:review` runs locally against the checkout instead, with no GitHub side effects. It is the same local `/review` that the GitHub-review section names as the fallback when an administrator has disabled Codex Cloud, reachable without leaving Claude Code. Its output stays in the terminal, so it does not create the durable review record that the lab's pull-request workflow relies on.

Useful to us? Yes, for local second-opinion reviews; not a replacement

The main use is `/codex:adversarial-review --base main` as a cross-vendor second opinion before pushing, which is what `ai-config`'s adversarial self-review rule asks for and what the `delegate-to-codex` skill implements by hand. The plugin makes that one command, and the `--background` flag keeps a multi-file review from blocking the session.

Three cautions apply:

- Leave the review gate off. A `Stop` hook that reruns Codex on every turn spends on both providers at the rate of a chat conversation, and the lab's own `Stop` hooks already gate on cheaper deterministic checks.
- `/codex:rescue` writes to the working tree by default. In a shared worktree or a multi-agent session, ask for a read-only run or use the `delegate-to-codex` skill, which sandboxes by default.
- The plugin is a wrapper, so it inherits Codex's model access rules: a ChatGPT login cannot reach every `--model` value the CLI accepts, and the refusal arrives from the API after the flag is accepted.

For fan-out work, structured output, or anything a script needs to consume, the `delegate-to-codex` skill remains the right tool. For pull-request reviews that must be visible to other

contributors, the GitHub integration in Codex GitHub review remains the right tool. See Section 13 for the rest of the plugins the lab has evaluated.

16 Anatomy of Agent Plugins

In modern AI coding assistants (such as Google Antigravity and Claude Code; see the harness landscape on the sunset of legacy Gemini CLI and its folding into Antigravity CLI), **plugins** serve as the top-level packaging and distribution layer for agent capabilities (measured 2026-09-01). While individual skills or Model Context Protocol (MCP) servers extend specific tasks, a plugin aggregates multiple extensibility primitives into a unified, version-controlled bundle.

Anatomy of a plugin bundle

A plugin manifest (such as `plugin.json` or `.claude-plugin/plugin.json`) orchestrates four distinct architectural components:

- **Skills (`skills/**/SKILL.md`)**: Procedural Markdown instructions that teach the agent domain-specific workflows, coding conventions, and structured checklists.
- **Model Context Protocol (MCP) servers**: External process definitions exposing executable tool functions, database connectors, and live workspace resources via standard MCP JSON-RPC endpoints.
- **Lifecycle hooks**: Deterministic executable scripts configured via hook manifests (such as `hooks.json`) attached to agent lifecycle events (such as `PreToolUse` command inspection, `Stop` review-gate verification, and `UserPromptSubmit` context injection).
- **Specialized agent roles and slash commands**: Pre-configured subagent personas (such as dedicated reviewers or researchers) and user-facing shortcut commands (`/command`).

Comparison: Skills vs. MCP Servers & Tools vs. Plugins

Dimension	Skills (`SKILL.md`)	MCP Servers & Tools	Plugins (`plugin.json`)
Primary purpose	Procedural guidance & workflows	External tool execution & data access	Unified packaging & distribution
Execution model	Progressively loaded on demand	Executed by agent harness over IPC	Discovered & loaded by agent platform
Dependencies	Plain Markdown & scripts	Language runtimes (Node.js, Python, binaries)	Bundles skills, MCP configs, and hooks

Dimension	Skills (`SKILL.md`)	MCP Servers & Tools	Plugins (`plugin.json`)
Lifecycle control	Passive context instructions	Dynamic tool calls during agent turn	Active deterministic hook gates

Managing token budget and context bloat

A common hazard when adopting large community plugin bundles is context window saturation. When multiple plugins eagerly inject verbose instructions and exhaustive tool schemas into every turn, available context for actual code and reasoning shrinks.

Effective plugin architectures mitigate this through several strategies:

- **Lazy tool discovery**: Only core system tools load eagerly; specialized plugin tools declare schemas that load lazily on demand.
- **On-demand skill activation**: Agents search skill catalogs dynamically when relevant keywords appear, rather than loading the entire skill directory into the initial system prompt.
- **Prefix caching preservation**: Static plugin definitions are placed at the root of prompt structures so provider-level prompt caching remains undisturbed during multi-turn sessions.

17 Setting up MCP servers

The Model Context Protocol (MCP) is how a harness gains typed access to external systems. Configuring a server is usually a one-line command. Diagnosing one that *silently* isn't working is the part worth writing down, because the common failure mode produces no error at all — only a quiet absence of tools you assumed were there.

> **Note**
>
> These notes were written in July 2026, from a real diagnosis on a Linux machine. As with the rest of this chapter, treat the specifics with caution — harness internals and vendor defaults change quickly.

Installed is not registered

A server binary sitting on disk is not available to your agent. The harness knows only what its configuration declares, so installing `github-mcp-server` and *registering* it are two separate acts. Skipping the second is easy to miss, because the first one felt like the hard part.

The check is one command:

```
claude mcp list
```

A broken server can occupy the name your working one wanted

This is the failure worth knowing about, and it is nastier than a missing entry, because `claude mcp list` shows you something that looks right.

Plugin marketplaces can register servers of their own. An official GitHub plugin may install a **remote** server under exactly the name you meant to give your **local** one. The listing then reports a `github` server that is not your setup at all, and the local binary you installed goes unregistered and unnoticed.

So read the listing for the *transport and address*, not just the name:

```
plugin:github:github: https://api.githubcopilot.com/mcp/ (HTTP) - X Failed to
connect - HTTP 400: ... Authorization header is badly formatted
```

An `(HTTP)` entry pointing at a vendor URL is a remote server. A local one shows a command path instead.

400 and 401 mean different things

The status code is the whole diagnosis here, and it is easy to skim past.

The configuration a plugin ships may hardcode a credential placeholder:

```
{"headers": {"Authorization": "Bearer ${GITHUB_PERSONAL_ACCESS_TOKEN}"}}
```

If that variable has no value when the harness starts, the header goes out as a bare `Bearer` with nothing after it. That is **malformed**, not **unauthorized**:

- **401** means the server read your token and rejected it — a real credential problem: expired, wrong scopes, wrong account.
- **400** means no token was ever substituted — a *configuration* problem, and no amount of re-issuing tokens will fix it.

Chasing a 400 as though it were a 401 sends you to the token-minting page for a problem that lives in a config file.

Install the binary the platform's own way

These notes started from a Linux install, where the binary lands under `$HOME/.local/bin`. On macOS the same server is a Homebrew formula, and getting it is one command:

```
brew install github-mcp-server
```

That installs to `/opt/homebrew/bin` on Apple silicon (`/usr/local/bin` under Intel Homebrew), so no manual download is needed.

The official install guide leads with a Docker recipe instead, and it has a catch worth knowing before you follow it: `docker` being on `PATH` does not mean the daemon is running. With Docker Desktop stopped, the server fails to connect to the Docker API, and that failure surfaces at *server start*, not at registration — so `claude mcp add` succeeds, and the break only shows up later, as a silent absence of tools. That is one more reason to prefer the binary path above.

Because the binary's location differs by platform and by installer, a launch wrapper should resolve it from `PATH` rather than hardcode it, with an override for the case where it isn't on one.

Supply credentials without storing a token

The obvious registration bakes a token straight into harness config:

```
claude mcp add github -e GITHUB_PERSONAL_ACCESS_TOKEN=<pat> -- <server> stdio
```

That writes a live credential to a config file in plain text, and pins you to one token that will eventually expire.

A launch wrapper avoids both. It reads the credential at start time from a tool that already holds one, so nothing is stored and the server follows whatever account you are currently logged in as:

```
#!/bin/sh
set -eu

SERVER="${GITHUB_MCP_SERVER_BIN:-$(command -v github-mcp-server || true)}"
if [ -z "$SERVER" ]; then
  echo "github-mcp-server not on PATH; install it or set GITHUB_MCP_SERVER_BIN" >&2
  exit 1
fi

GITHUB_TOOLSETS="${GITHUB_TOOLSETS:-default,actions}"
```

```
export GITHUB_TOOLSETS

GITHUB_PERSONAL_ACCESS_TOKEN="$(gh auth token)"
if [ -z "$GITHUB_PERSONAL_ACCESS_TOKEN" ]; then
  echo "empty token; run 'gh auth login'" >&2
  exit 1
fi
export GITHUB_PERSONAL_ACCESS_TOKEN

exec "$SERVER" stdio "$@"
```

Register the wrapper rather than the binary:

```
claude mcp add --scope user github -- ~/.local/bin/github-mcp-server-stdio
```

Note the explicit failure when the token comes back empty. A wrapper that silently exports an empty string reproduces the bare-`Bearer` bug you just finished diagnosing.

Toolsets are opt-in, and the default may omit what you need

A server does not necessarily expose everything it can do. GitHub's server exposes a default group, and that default carries **no continuous-integration access at all** — no workflow runs, no job logs, no re-run trigger. As of this writing (server v1.7.0) the actions toolset's four tools are `actions_get`, `actions_list`, `actions_run_trigger`, and `get_job_logs` — the exact names have moved around across releases, so treat this list as a snapshot rather than a promise.

If your workflow involves driving pull requests to a clean state, that omission matters, because reading check status is most of the job. Request the extra group explicitly:

```
GITHUB_TOOLSETS=default,actions
```

Note that the selection **replaces** the default rather than extending it, which is why the value above names `default` explicitly. Writing `actions` alone would silently trade away every default tool for the four you asked for — a net *loss* that looks like a successful configuration change.

So compare the tool list before and after, and confirm the count went up rather than sideways. On the same v1.7.0 server, the default group carries 44 tools, and `default,actions,discussions,dependabot,labels,notifications` carries 63 — the count moving the right direction, as the check above expects.

`subscribe_pr_activity` isn't a local-server tool

Workflow guidance written for remote or web agent sessions sometimes names `subscribe_pr_activity` as the way to watch a pull request's activity. It doesn't appear in a locally-run GitHub MCP server, under any toolset combination.

The local analogues are `manage_notification_subscription` and `manage_repository_notification_subscri` Reach for those instead when working from a local harness.

Verify with a real call, and expect to restart

Two habits close this out.

Verify by calling, not by reading a list. A tool appearing in the registry proves the harness parsed a config file. It does not prove the server started, authenticated, or can reach the API. One cheap identity call plus one read (for GitHub: `get_me`, then listing pull requests on a repo you know) proves the whole path end to end.

Expect the tools to be missing until you restart. MCP servers connect when a session starts, so a server registered mid-session is inert for the rest of it. This is a common false alarm: the registration worked, and the tools genuinely are not there yet.

Finally, note which new tools can *write*. A re-run or dispatch tool can trigger CI, and permissive permission modes will not prompt before it does. Treat those the way you would treat a merge — something a human authorizes, not something an agent does in passing.

Copilot on GitHub uses a different config surface

The notes above are for a local harness (`claude mcp list`, a binary on `PATH`). Copilot cloud agent and Copilot code review on GitHub.com do not read that file.

Repository administrators configure those agents from **Settings > Copilot > MCP servers** using a JSON `mcpServers` object. GitHub's Configure MCP servers page is the source of truth for the schema. As of that page:

- The GitHub MCP server and the Playwright MCP server are enabled by default.
- Cloud agent and code review share the repository config; a separate toggle can disable MCP tools for code review only.
- Only MCP *tools* are supported, not resources or prompts.
- Remote servers that authenticate with OAuth are not supported.
- Secrets and variables must be named with a `COPILOT_MCP_` prefix or they are invisible to the config.
- Once a tool is enabled, Copilot uses it without asking for approval, so allowlist specific read-only tools rather than `*`.

Do not copy a local `claude mcp add` registration into that JSON and expect it to work.

Granola MCP: your meetings as context

Granola MCP is the other side of that same pattern: it is not a code-host server but a meeting-context server. Granola is an AI notepad for back-to-back meetings; its MCP exposes your meeting notes to any MCP client.

The gap it closes is the copy-paste loop: without it, using something you discussed in a meeting while working in Claude, ChatGPT, or Cursor means finding the note, copying the relevant bit, and pasting it in. With the MCP connected, that context rides with you. Use it to turn a standup into Linear tickets, scaffold a feature from what was agreed, or draft a follow-up from what was actually said.

It connects through the standard Model Context Protocol (MCP) as a remote server at `https://mcp.granola.ai/mcp`. For Claude or ChatGPT, enable it from the app's connector/app settings and authenticate; for Cursor, Claude Code, or any other MCP client that supports a manual URL, register that URL directly (see the announcement for per-client steps). On an Enterprise plan it is an early-access beta, off by default until an admin enables it.

18 Google Antigravity Python SDK

The `google-antigravity/antigravity-sdk-python` repository provides the official Python SDK for building and automating agents on the Google Antigravity agent runtime (measured 2026-08-31; distributed via PyPI as `google-antigravity`).

Architecture and runtime model

The SDK embeds the compiled Antigravity runtime engine directly into Python applications:

- **Lifecycle management**: Agents are instantiated through `Agent` objects configured via `LocalAgentConfig`, managed within asynchronous Python context managers (`async with`).
- **Autonomous agentic loop**: The underlying runtime drives multi-turn reasoning, streaming model responses, subagent spawning, and tool dispatch without requiring hand-rolled state machines.
- **Binary distribution**: The Python package distributes pre-compiled native runtime binaries via wheels, ensuring consistent agent execution across Windows, macOS, and Linux environments.

Extensibility and policy enforcement

Developers can customize agent behavior and enforce safety policies:

- **Custom tools and MCP servers**: Register custom Python functions as callable agent tools or connect external Model Context Protocol (MCP) server endpoints.
- **Steering hooks**: Attach pre-tool and post-tool lifecycle hooks to inspect, steer, or veto actions (such as restricting command execution or gating repository modifications).
- **Skill discovery**: Load skill catalogs (`skills/**/SKILL.md`) dynamically, allowing agents to leverage shared procedural instructions.

Comparison with interactive Antigravity surfaces

Dimension	Antigravity Python SDK	Antigravity CLI & IDE Extensions
Primary use case	Automated pipelines, CI evaluation, custom harnesses	Interactive terminal and GUI pair programming
Control plane	Python API (`async with Agent(...)`)	Interactive CLI prompt or editor chat panel
Tool definitions	In-process Python callables & MCP	JSON manifests, plugins, and CLI scripts
Runtime engine	Embedded native binary	Managed local service

19 Unbounded Context with Magic Context

cortexkit/magic-context is an open-source self-managing memory engine designed to provide unbounded context for AI coding agents (measured 2026-08-31). It operates as a background memory subsystem—often described as a "hippocampus for coding agents"—that extracts, consolidates, and retrieves long-term repository state without pausing the active coding turn.

Core architecture and agent roles

Rather than requiring the primary coding agent to interrupt its execution to prune conversation buffers, `magic-context` delegates memory lifecycle operations to specialized background workers:

- **The Historian**: Runs background context compaction on completed turns, compressing verbose tool outputs and dialog history while preserving architectural decisions and code rationale.

- **The Dreamer**: Executes periodic consolidation passes to deduplicate memory records across multiple sessions, distilling recurring observations into canonical project facts and persistent guidelines.
- **The Sidekick**: Acts as an on-demand retrieval companion that augments active prompts with relevant project context, supplying historical context matched to the current file or task.

Cache-aware memory management

A key challenge with dynamic prompt injection is preserving prompt caching efficiency. `magic-context` addresses this through cache-conscious orchestration:

- **Cache-stable prompt layout**: Maintains a deterministic prompt layout and replay ordering, preserving provider prompt caching prefixes across conversational turns without invalidating cached tokens.
- **Deferred background extraction**: Memory analysis and summarization tasks are deferred to idle windows or subagent threads, preventing token churn and latency spikes during high-tempo coding loops.
- **Cross-session persistence**: Extracted knowledge persists in lightweight local stores across IDE restarts, enabling coding agents to resume work with full institutional memory of past decisions.

20 Spec-Driven Development with Conductor

gemini-cli-extensions/conductor is an open-source plugin for AI coding agents (including Google Antigravity and Claude Code) that implements **Spec-Driven Development** (measured 2026-08-31). Rather than relying on conversational chat history that degrades over extended sessions, Conductor anchors agent behavior in structured, version-controlled Markdown artifacts stored directly in the repository, providing persistent context across multi-session workflows.

Core workflow phases

Conductor structures development into four distinct, sequential phases:

- **Context establishment (`/conductor:conductor-setup`)**: Interactively initializes baseline project documentation (including product goals, technical stack choices, and testing guidelines), giving coding agents persistent reference material across subsequent sessions.
- **Specification and track planning (`/conductor:conductor-new-track`)**: Transforms feature requests or bug fixes into a dedicated track containing a `spec.md` (functional scope and acceptance criteria) and a `plan.md` (ordered implementation phases broken into verifiable task checklists).

- **Phased implementation (`/conductor:conductor-implement`)**: Guides the agent through the active track's plan sequentially, executing file edits, running local test suites, and marking tasks complete as acceptance criteria are met.
- **Track review and plan compliance (`/conductor:conductor-review`)**: Conducts an adversarial verification pass against the original track specification, ensuring that all declared acceptance criteria are satisfied and no architectural drift occurred during execution.

Architectural benefits of Spec-Driven Development

Dimension	Conversational Prompting	Spec-Driven Development (Conductor)
Context persistence	Volatile in-memory chat buffer	Version-controlled Markdown artifacts
Task boundaries	Ad-hoc user instructions per turn	Structured `spec.md` and `plan.md` checklists
Verification loop	Manual spot-checking	Milestone-level automated tests and `/conductor:conductor-review`
Handoff & resumption	Requires re-prompting or context replay	Any agent resumes from the checked-in track state

21 Managing Gemini API Spend and Cost Optimization

This guide describes how to manage Google AI Studio and Google Cloud Gemini API spend caps, unpause paused API services, and optimize token consumption across local tools and GitHub Actions workflows.

Identifying Paused Projects (Project Numbers vs. Friendly Names)

When a project reaches its monthly budget limit, Google sends an email notification stating that Gemini API service has been paused. These email notifications identify the affected project using its internal **numerical GCP Project Number** (e.g. `156839315029`).

In contrast, Google AI Studio lists projects by their **friendly display names** (such as `ai-config Project` or `gha-project`) and alphanumeric client IDs (`gen-lang-client-...`).

To find which project in AI Studio corresponds to the notification email:

1. Open Google AI Studio Projects.

2. Locate the project corresponding to your client ID or spend alert.
3. Any project that has hit its monthly spend cap will have its API requests paused until the limit is updated.

Adjusting and Unpausing Spend Caps

To restore API access for a paused project:

1. Open Google AI Studio Spend.
2. Increase the monthly spend cap dollar amount or set it to unlimited.
3. Save your changes. API requests will automatically resume within a few minutes.

If no manual action is taken, accumulated spend resets to **$0 on the 1st of the next month**, and API service automatically resumes up to the configured cap.

Setting Up Google Cloud Budget Alerts

To receive early warnings before reaching a spend cap:

1. Open Google Cloud Console Billing Budgets & Alerts.
2. Select your billing account and click **Create Budget**.
3. Name the budget (e.g., `Lab AI API Monthly Budget`) and select the relevant GCP projects.
4. Set your target monthly budget amount.
5. Configure trigger rules for email notifications at **50%**, **75%**, and **90%** of the budget.
6. Click **Finish**. You will receive email alerts before any project hits its spend cap.

Cost Optimization Best Practices

To maximize the efficiency of your API spend across local CLI sessions, subagents, and automated workflows:

- **Right-Size Model Selection**: For general Gemini API, Python SDK, or custom scripts, prefer Flash-tier models (such as `gemini-2.5-flash`) over Pro-tier models (`gemini-2.5-pro`). Flash models provide a substantially lower token cost for routine search, log parsing, and background processing. For Antigravity Agent workflows (`google-antigravity`), the agent defaults to `gemini-3.7-flash` (already a Flash-tier model). Supported Antigravity Agent model options include `gemini-3.7-flash`, `gemini-3.6-flash`, `gemini-3.5-flash`, and `gemini-3.5-flash-lite`.
- **Use Context Caching**: For workloads that repeatedly pass static context over 2,048 tokens (such as large reference docs, system prompts, or codebase indices), use Gemini Context Caching. Cached input tokens receive a substantial discount compared to standard input tokens.

- **Use the Batch API for Non-Realtime Tasks**: For offline batch processing, evaluation suites, or background doc updates, submit requests via the Gemini Batch API to receive a 50% discount on input and output tokens.
- **GitHub UI Diff Collapsing**: Mark dependency lockfiles (`*.lock`, `package-lock.json`, `yarn.lock`, `renv.lock`) and generated build artifacts as `linguist-generated=true` in `.gitattributes` to collapse them in GitHub's web diff view and exclude them from repository language statistics.

22 Claude Code Cloud Environments

Claude Code is a CLI coding agent that can also run tasks on Anthropic-managed cloud infrastructure— either from the web at claude.ai/code ("Claude Code on the web"), or from the terminal by adding the `--remote` flag to move a session into the cloud.

Each cloud run executes inside a configured **environment**. An environment bundles three things:

- **Network access level**: what the cloud session is allowed to reach (a security control, analogous to the firewall configuration described above).
- **Environment variables**: values the session needs at runtime, such as `NODE_ENV`, database URLs, or API keys.
- **Setup scripts**: Bash that runs automatically when the session starts, for example to install dependencies.

Selecting the default environment with `/remote-env`

The `/remote-env` slash command sets **which configured environment is the default** for `--remote` runs:

- With a single environment configured, it shows your current configuration.
- With multiple environments, it opens an interactive picker so you can choose the default.

`/remote-env` only *selects* the default; to add, edit, or archive the environments themselves, use the web interface at claude.ai/code. Because `/remote-env` opens an interactive panel, run it from an interactive `claude` terminal session.

> **Availability**
>
> Claude Code on the web (and the cloud environments it relies on) is a research-preview feature, available to Pro, Max, and Team users (and Enterprise users with eligible seats). Availability and behavior may change.

For details, see the Claude Code on the web documentation and the slash command reference.

References

Anthropic. 2026a. *Claude Code Plugins Directory*. Software. https://github.com/anthropics/claude-code/tree/main/plugins.

Anthropic. 2026b. *Claude Code Plugins Official Marketplace*. Software. https://github.com/anthropics/claude-plugins-official.

Anthropic. 2026c. *Claude Plugins Community Marketplace*. Software. https://github.com/anthropics/claude-plugins-community.

Anthropic. 2026d. *Discover and Install Prebuilt Plugins Through Marketplaces*. Documentation. https://code.claude.com/docs/en/discover-plugins.

Anthropic. 2026e. *Ralph Loop Plugin*. Documentation. https://raw.githubusercontent.com/anthropics/claude-plugins-official/main/plugins/ralph-loop/README.md.

Coles, Matt. 2026. *Herding Parallel Agents on a Remote Box with Herdr*. Blog post. https://coles.codes/posts/herding-agents-with-herdr/.

Copes, Flavio. 2026. *A Deep Dive into Herdr*. Blog post. https://flaviocopes.com/herdr/.

Databricks. 2026a. *Foundation Model APIs Limits and Regions*. Documentation. https://docs.databricks.com/aws/en/machine-learning/foundation-model-apis/limits.

Databricks. 2026b. *Supported Models in Foundation Model APIs*. Documentation. https://docs.databricks.com/aws/en/machine-learning/foundation-model-apis/supported-models.

Hacker News. 2026. *Herdr: Agent Multiplexer That Lives in Your Terminal*. Show HN discussion thread. https://news.ycombinator.com/item?id=48714802.

herdrdev. 2026a. *Agents*. Herdr documentation. https://herdr.dev/docs/agents/.

herdrdev. 2026b. *Concepts and Keybindings*. Herdr documentation. https://herdr.dev/docs/concepts/.

herdrdev. 2026c. *Connecting Machines*. Herdr documentation. https://herdr.dev/docs/connecting-machines/.

herdrdev. 2026d. *Herdr Agent Guide*. Herdr documentation. https://herdr.dev/agent-guide.md.

herdrdev. 2026e. *Herdr: The Runtime Coding Agents Run on*. Website. https://herdr.dev/.

herdrdev. 2026f. *Herdrdev/Herdr: The Runtime Your Coding Agents Live on*. GitHub repository. https://github.com/herdrdev/herdr.

herdrdev. 2026g. *Install Herdr*. Herdr documentation. https://herdr.dev/docs/install/.

herdrdev. 2026h. *Session State*. Herdr documentation. https://herdr.dev/docs/session-state/.

herdrdev. 2026i. *Socket API*. Herdr documentation. https://herdr.dev/docs/socket-api/.

Huntley, Geoffrey. 2025. *Ralph*. Blog post. https://ghuntley.com/ralph/.

Konur, Yigit. 2026. *Awesome-Herdr: A Curated Guide to the Herdr Ecosystem*. GitHub repository. https://github.com/yigitkonur/awesome-herdr.

Morrison Lab. 2026a. *Ai-Config: Portable AI Agent Config*. Software. https://github.com/Morrison-Lab/ai-config.

Morrison Lab. 2026b. *Delegate-to-Codex: Run Heavy Sidecar Work on Codex, Not Claude*. Documentation. https://github.com/Morrison-Lab/ai-config/blob/main/skills/delegate-to-codex/SKILL.md.

OpenAI. 2026a. *Codex Plugin for Claude Code*. Software. https://github.com/openai/codex-plugin-cc.

OpenAI. 2026b. *Codex Pricing*. Documentation. https://developers.openai.com/codex/pricing.

Posit PBC. 2026a. *Ai-Lib: LLM Provider Infrastructure for Posit Assistant*. Software. https://github.com/posit-dev/ai-lib.

Posit PBC. 2026b. *Assistant-Feedback: Repository for Feedback on Posit Assistant*. Software. https://github.com/posit-dev/assistant-feedback.

Posit PBC. 2026c. *Bluffbench2: A Visual Reasoning LLM Benchmark*. Software. https://github.com/posit-dev/bluffbench2.

Posit PBC. 2026d. *Btw: A Complete Toolkit for Connecting r and LLMs*. Software. https://github.com/posit-dev/btw.

Posit PBC. 2026e. *Chatlas: Your Friendly Guide to Building LLM Chat Apps in Python*. Software. https://github.com/posit-dev/chatlas.

Posit PBC. 2026f. *Code-Index: Semantic Code Search for AI Coding Assistants*. Software. https://github.com/posit-dev/code-index.

Posit PBC. 2026g. *Commons: AI Agents for Data Analysis*. Software. https://github.com/posit-dev/commons.

Posit PBC. 2026h. *Mcp-Repl: A Persistent, Sandboxed r or Python Session for MCP Agents*. Software. https://github.com/posit-dev/mcp-repl.

Posit PBC. 2026i. *Mcptools: Model Context Protocol for r*. Software. https://github.com/posit-dev/mcptools.

Posit PBC. 2026j. *Nesevals: Evaluate Next Edit Suggestion Scaffolds*. Software. https://github.com/posit-dev/nesevals.

Posit PBC. 2026k. *Positron, a Next-Generation Data Science IDE*. Software. https://github.com/posit-dev/positron.

Posit PBC. 2026l. *Positron-Copilot-Chat: Copilot Chat Extension for Positron*. Software. https://github.com/posit-dev/positron-copilot-chat.

Posit PBC. 2026m. *Querychat: Chat with Your Data in r and Python*. Software. https://github.com/posit-dev/querychat.

Posit PBC. 2026n. *Raghilda: RAG Made Simple*. Software. https://github.com/posit-dev/raghilda.

Posit PBC. 2026o. *Shiny-Assistant: Chat Assistant for Shiny*. Software. https://github.com/posit-dev/shiny-assistant.

Posit PBC. 2026p. *Shinychat: Chat UI Component for Shiny*. Software. https://github.com/posit-dev/shinychat.

Posit PBC. 2026q. *Shinyrealtime: OpenAI Realtime API for Shiny*. Software. https://github.com/posit-dev/shinyrealtime.

Tessl. 2026a. *Core Concepts*. Documentation. https://docs.tessl.io/introduction-to-tessl/core-concepts.

Tessl. 2026b. *Installation*. Documentation. https://docs.tessl.io/introduction-to-tessl/set-up-tessl/installation.

Tessl. 2026c. *Protecting Yourself from Insecure Skills*. Documentation. https://docs.tessl.io/

tutorials/protecting-against-insecure-skills.

Tessl. 2026d. *Sharing Usage Data*. Documentation. https://docs.tessl.io/legal/sharing-usage-data.

Tessl. 2026e. *Supported Platforms*. Documentation. https://docs.tessl.io/support/supported-platforms.

Tessl. 2026f. *Tessl FAQs*. Documentation. https://docs.tessl.io/support/faqs.

Tessl. 2026g. *Tessl Pricing*. Website. https://tessl.io/pricing.

Tessl. 2026h. *Tessl Skills Registry*. Website. https://tessl.io/registry.

Tessl. 2026i. *Tessl: Agent Enablement Platform*. Website. https://tessl.io/.

Tessl. 2026j. *Using Tessl as a Package Manager*. Documentation. https://docs.tessl.io/tutorials/using-tessl-as-a-package-manager.

Tessl. 2026k. *What Is Tessl?* Documentation. https://docs.tessl.io/.

Vincent, Jesse. 2026. *Superpowers*. Software. https://github.com/obra/superpowers.

Y Combinator. 2026. *Herdr: Building the Open Agent Runtime*. Company profile. https://www.ycombinator.com/companies/herdr.