ANTHROPIC CLAUDE & WORKSPACE STUDIOS • FIELD GUIDE

Claude Code Mods

Customizing Claude Code from the inside with in-process TypeScript and JavaScript mods—building live terminal HUDs, prompt and tool interceptors, custom slash commands, and `sec-default` guardrails.

In-Process TS/JS Engine | `register(on, options)` | Before / After / Around Hooks

Custom Terminal & App HUDs | Hot-Reload `dev-mods` | `sec-default` Governance



Middleware Agent Control

Intercept, rewrite, time, or short-circuit prompts and tool calls in memory via Express-style `($, e, next)` pipelines.

Programmable Terminal UI

Mount live token gauges, cache TTL timers, and side panes across both the CLI and Claude Code Desktop using `ui.render`.

Zero-Restart Hot Reload

Describe a workflow tweak in plain English and watch Claude Code build and hot-reload session `dev-mods` instantaneously.

Why Claude Code Mods Change Everything

Classic shell hooks spawn slow external subprocesses that can only say yes or no, while MCP servers sit outside the agent waiting to be called. Claude Code Mods run *inside* the live process as native TypeScript and JavaScript middleware—letting you reshape prompts, rewrite tool calls, and paint custom UI frames at sub-millisecond speed.

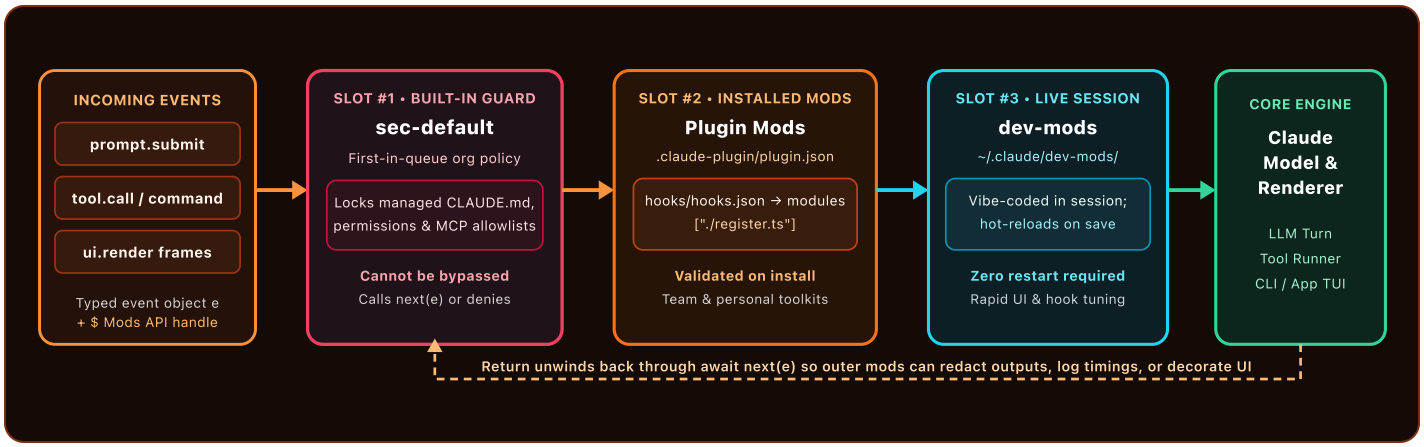
Quick Navigation

1. Middleware Mental Model & Queue SVG p. 2
2. Setup, File Layout & 4-Way Matrix p. 3
3. Annotated Mod-Enhanced UI Tour p. 4
4. Anatomy of `register(on, options)` p. 5
5. The `$` API, Zero-NPM & `/commands` p. 6
6. Vibe-Modding & Live `dev-mods` p. 7
7. Custom UI, HUDs & Panel Replacement p. 8
8. 5 Real-World Copy-Paste Mod Recipes p. 9
9. Security, Load Order & `sec-default` p. 10
10. Cross-Tool Synergy Pipeline p. 11
11. Troubleshooting Clinic & Pre-Flight p. 12
Golden Rule: Zero NPM, Always Return `next` All

From External Glue Scripts to In-Process Agent Middleware

Think of Claude Code's runtime loop like an HTTP web server, except the packets flowing through it are **user prompts**, **tool invocations**, **slash commands**, and **UI render frames**. Before Mods, customizing Claude Code meant bolting shell scripts onto `settings.json`—paying Node or Python startup overhead on every tool call, losing in-memory state between turns, and having zero control over the terminal screen. Mods replace brittle glue scripts with an ordered chain of in-process TypeScript functions that share persistent session state and pass events forward via `await next(e)`.

Sequential Middleware Pipeline: From Raw Event to Engine & Back



Why the Unwind Path Matters Just as Much as the Forward Pass

Because every handler wraps `await next(e)` like peeled layers of an onion, a single 15-line mod can record a timestamp *before* a tool runs, let the tool execute, and scrub leaked `.env` tokens from the tool's stdout *before* those bytes ever enter Claude's context window.

Prerequisites, File Layout & Extension Matrix

A Claude Code Mod is simply a standard Claude Code plugin whose `hooks/hooks.json` declares a `"modules"` array. There is no build step, bundler, or `tsconfig.json` required—Claude Code compiles and validates your TypeScript natively at load time.

1 Runtime Prerequisites

- **Claude Code** `v2.1.287+`**:** Minimum release supporting in-process hook `"modules"`, the `$` Mods API, and `ui.render` across CLI and Desktop.
- **Zero Build Chain Needed:** Native `.ts` and `.js` ES modules load directly without Webpack, Vite, or `node_modules`.
- **Built-In Validator:** Run `claude plugin validate ./my-mod` to trigger static capability scanning before enabling any mod.

2 Minimal 3-File Mod Bundle

```
my-guardrail-mod/
├── .claude-plugin/
│   └── plugin.json       # Name, version & optional userConfig
└── hooks/
    ├── hooks.json        # Declares "modules": ["./register.ts"]
    └── register.ts       # Exports register(on, options)
```

Paths inside `"modules"` resolve relative to the `hooks/` folder.

```
# 1. Confirm Claude Code meets the v2.1.287+ Mods runtime floor:
claude --version                                  # Must output 2.1.287 or newer

# 2. Run static capability & schema validation on a local mod directory before loading:
claude plugin validate ./my-guardrail-mod         # Audits $ API usage, blocks npm imports & prints capabilities
```

Ecosystem Decision Matrix: Mods vs. Shell Hooks vs. Skills vs. MCP

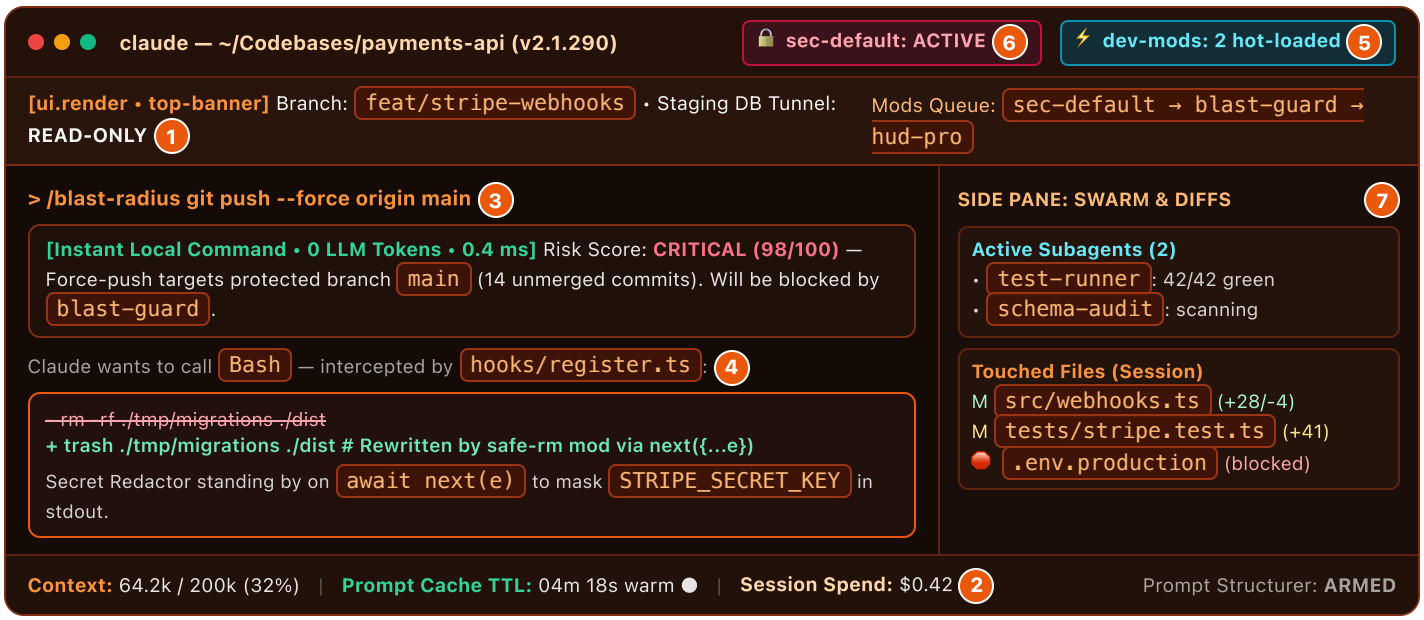
Dimension	Claude Code Mods (`register.ts`)	Classic Shell Hooks (`settings.json`)	Claude Skills (`SKILL.md`)	MCP Servers (Stdio / HTTP)
Execution Model & Latency	**In-process TS/JS** (<1 ms); persistent memory across the whole session.	Spawns a fresh OS subprocess per event (50–400 ms startup overhead).	Markdown instructions loaded into prompt context on demand.	Out-of-process RPC server invoked only when Claude picks a tool.
Prompt & Tool Rewriting	**Full read/write middleware:** mutate inputs, mock results, or redact outputs.	Mostly allow/block exit codes and limited JSON stdout flags.	Guides model reasoning softly; zero deterministic enforcement.	Cannot see or intercept prompts or built-in `Bash` / `Edit` calls.
Terminal & App UI Control	**Native** `ui.render`**:** status bars, HUD bands, side panes & panel overrides.	None—prints plain text logs only on failure or transcript dump.	None—purely conversational behavior and domain templates.	None—returns structured tool payloads back to the model.
External `npm` **& SDKs**	**Disallowed:** zero `npm`/WASM so static scanner can audit capabilities.	Allowed (any local binary, Python script, or shell tool).	Can bundle helper scripts executed via standard shell tools.	**Best choice** for heavy SDKs, DB drivers, OAuth & external APIs.
Security Boundary	Runs in-process with user privileges; governed by `$` scanner + `sec-default`.	Runs with user shell privileges whenever matching event fires.	Prompt-level text; relies on tool permissions for execution.	Isolated process; gated per-tool by Claude Code permissions.

Quick Selection Rule: When to Reach for a Mod Instead of an MCP Server

Build an **MCP server** when Claude needs a *new external capability* that requires heavy libraries (like querying Postgres or Stripe). Build a **Claude Code Mod** when you want to *reshape how Claude Code itself behaves or looks*—enforcing deterministic guardrails, rewriting prompts, adding instant zero-token slash commands, or mounting live telemetry HUDs.

Annotated Tour: Mod-Enhanced Claude Code Cockpit

Mods transform Claude Code from a scrolling text stream into an instrumented cockpit. Below are the seven live touchpoints where TypeScript mods render telemetry, intercept risky commands, and expose instant local controls.



Interface Legend: The 7 Mod-Powered Touchpoints

1 **Environment Banner (`ui.render`):** Persistent header painting live Git branch, active cloud profile, and ordered mod queue so you never run prod commands by mistake.

2 **Prompt-Band Telemetry HUD:** Draws right above the input prompt with context fullness, 5-minute prompt-cache TTL countdown, and live session cost.

3 **Zero-Token Slash Command (`command.run`):** Local commands like `/blast-radius` execute pure TypeScript in under a millisecond without spending a single LLM token.

4 **Tool Call Rewriter (`tool.call`):** Inspects `Bash` or `Edit` arguments before execution—rewriting `rm -rf` to `trash` or denying edits to protected paths.

5 **Live `dev-mods` Hot-Reload Badge:** Shows active session mods loaded from `~/.claude/dev-mods/<session-id>/` that rebind automatically on file save.

6 **`sec-default` Baseline Lock:** Confirms Anthropic's built-in org policy mod occupies Slot #1, shielding managed rules and enterprise permissions.

7 **Docked Side Pane & Panel Override:** Uses side-pane and diff render slots in wide terminals and Claude Code Desktop to track subagent tasks, line-count diffs, and blocked file touches without scrolling up.

Anatomy of a Mod: `register(on, options)`

Every mod exports a single `register(on, options)` function. Using `on(eventName, filter, handler)`, your handler receives `($, e, next)`—giving you four clean middleware patterns: **before**, **instead of**, **after**, and **around**.

1. `.claude-plugin/plugin.json`

```
{
  "name": "ops-shield",
  "version": "1.0.0",
  "description": "Prompt cleaner, Bash guard & secret
redactor",
  "userConfig": {
    "strictGit": { "type": "boolean", "default": true }
  }
}
```

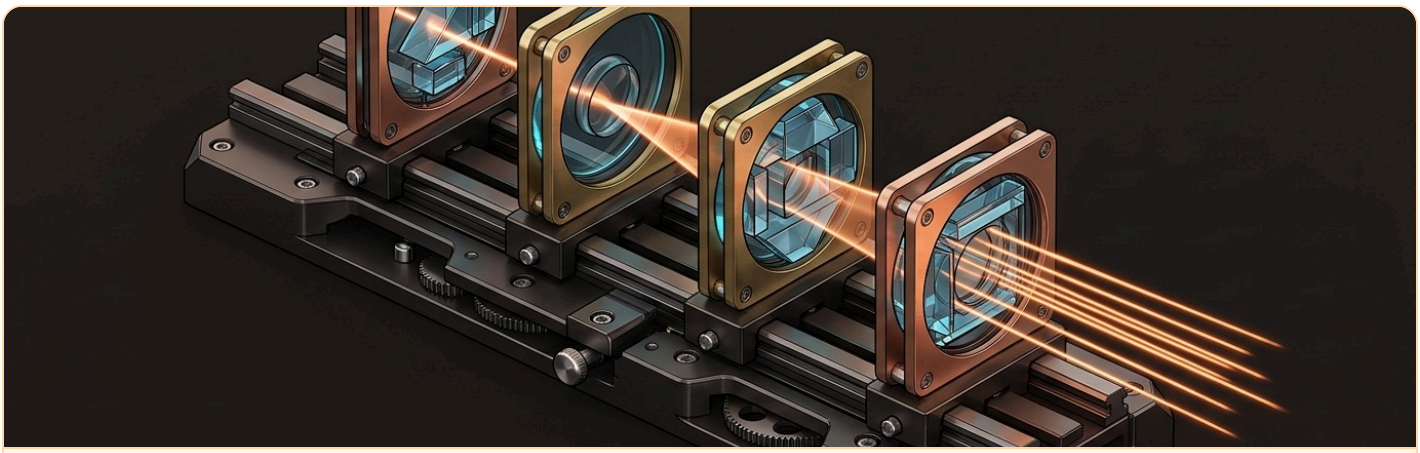
2. `hooks/hooks.json`

```
{
  "description": "In-process middleware entry module",
  "modules": [
    "./register.ts"
  ]
}
// No shell subprocesses — loads directly into the engine
```

3. `hooks/register.ts` — Before, Instead-Of, After & Around Middleware (Zero External Imports)

```
export function register(on: any, options: { strictGit?: boolean } = {}) {
  // 1. BEFORE (prompt.submit): Rewrite messy voice/brain-dump prompts before Claude sees them
  on('prompt.submit', async ($: any, e: any, next: any) => {
    const raw = (e.prompt || '').trim();
    if (raw.startsWith('fix:')) {
      const structured = raw.slice(4).trim() + '\n\nConstraints: Run existing tests first and show a minimal diff.';
      return next({ ...e, prompt: structured });
    }
    return next(e);
  });

  // 2. INSTEAD OF + AROUND + AFTER (tool.call on Bash): Block destructive commands & redact .env secrets
  on('tool.call', { tool: 'Bash' }, async ($: any, e: any, next: any) => {
    const cmd = String(e.input?.command || '');
    if (cmd.includes('rm -rf /') || (options.strictGit !== false && cmd.includes('git reset --hard'))) {
      $.ui.toast({ level: 'warning', message: 'Blocked destructive command: ' + cmd });
      return { deny: 'Blocked by ops-shield mod. Use git stash or targeted file removal instead.' };
    }
    const startedAt = Date.now();
    const result = await next(e); // Execute downstream mods + actual Bash tool
    const elapsedMs = Date.now() - startedAt;
    const cleanOut = String(result?.output || '').replace(/(API_KEY|SECRET|TOKEN)=([^\s]+)/gi, '$1=[REDACTED]');
    await $.state.set('lastBashMs', elapsedMs);
    return { ...result, output: cleanOut };
  });
}
```



Four Interception Modes in One Pipeline: Each copper lens gate can mutate inputs *before* calling `next({ ...e })`, short-circuit *instead of* execution with `{ deny }`, scrub secrets *after* `await next(e)` resolves, or measure latency *around* the entire round trip.

The Mods API (`$`), Zero-NPM Rule & `/commands`

Inside a mod, standard Node globals like `require('fs')`, `process.env`, and external `npm` packages are intentionally off-limits. Instead, every host action flows through the structured `$` **Mods API** so Claude Code can statically audit what a plugin touches before it ever runs.

Core Namespaces Inside the `$` Capability Handle

`$.ui` & Layout

Trigger non-blocking `$.ui.toast()` alerts, request re-renders, and return declarative box/row/badge trees inside `ui.render` slots.

`$.state` & `$.timer`

Persist typed key-value counters across turns (`$.state.get/set`) and schedule clean session-scoped countdown intervals.

`$.fs` & `$.process`

Read workspace metadata or run quick helper commands (like `git status --porcelain`) through auditable host methods.

`$.command`

Register custom autocomplete slash commands on `session.start` with argument hints and instant local execution.

`$.model`

Run lightweight sidecar completions (e.g., Haiku classification or prompt cleanup) without polluting the main chat transcript.

`$.session`

Inspect token usage, active model tier, working directory, and cache timestamps to power real-time status gauges.

Why External `npm` Packages, React & WASM Binaries Are Blocked

When you run `claude plugin validate` or install a third-party mod, Claude Code's **static capability scanner** parses the abstract syntax tree of `register.ts` to list every `$` namespace and event hook the mod uses (for example: *reads files via* `$.fs` *, intercepts* `tool.call` *, makes zero network calls*). Allowing opaque `node_modules` trees, React reconcilers, or compiled `.wasm` blobs would blind static analysis and hide supply-chain attacks. Keep mods pure, lean TypeScript—and delegate heavy SDK work to an MCP server.

Building Zero-LLM-Latency Slash Commands with `command.run`

Markdown slash commands send a prompt to Claude and wait seconds for token generation. Mod-registered commands intercept `command.run` locally and return instant results in under a millisecond at **zero token cost**:

```
export function register(on: any) {
  // Register /checkpoint in Claude Code's slash-command autocomplete menu on startup
  on('session.start', async ($: any, e: any, next: any) => {
    await $.command.register({
      name: 'checkpoint',
      description: 'Show dirty Git files and token budget instantly without calling the model',
      argumentHint: '[optional-label]'
    });
    return next(e);
  });

  // Handle /checkpoint locally in sub-millisecond TypeScript — zero LLM round-trip!
  on('command.run', { command: 'checkpoint' }, async ($: any, e: any) => {
    const gitOut = await $.process.exec('git', ['status', '--short']);
    const stats = await $.session.stats();
    const label = e.args ? ' [' + e.args + ']' : '';
    const summary = gitOut.stdout.trim() || 'Working tree clean.';
    return {
      text: 'Checkpoint' + label + ' (' + stats.contextTokens + ' tokens used):\n' + summary
    };
  });
}
```

Hybrid Pattern: Local Fast-Path First, Optional Sidecar Model Second

Inside `command.run` , handle deterministic checks (Git state, port checks, regex lint) in pure TypeScript first, and call `$.model.complete()` only when genuine natural-language summarization is needed.

“Vibe-Modding”: Live Hot-Reload Without TypeScript Mastery

You never have to memorize event schemas to build a mod. Describe the behavior you want in plain English right inside an active Claude Code session: Claude writes the TypeScript into `~/.claude/dev-mods/<session-id>/`, validates it, and hot-reloads it on save with zero restart.

1 Ask in Plain English

Tell Claude Code what event to intercept or what HUD widget to draw. Built-in plugin-authoring context supplies exact `$` types automatically.

2 Instant Hot-Reload

Claude writes to `~/.claude/dev-mods/<session-id>/<mod>/`. A file watcher swaps handlers in memory immediately while keeping your chat alive.

3 Promote to Plugin

Once a session mod feels indispensable, ask Claude to package it into your repo’s plugins folder and run `claude plugin validate`.

Mod Authoring Prompt Anatomy: The 5 Color-Coded Building Blocks

Whether you are prompting Claude Code to generate a `dev-mod` or writing a spec for your team, include these five clauses so the generated handler passes static validation on the first try:

[1. Target Event] [2. Interception Mode] [3. UI Surface] [4. Zero-NPM Constraint] [5. Safety Fallback]

“Build a live session dev-mod named `migration-guard`. Listen to tool.call for Write and Edit, plus session.start for a /lock-db toggle command. Intercept before execution: if any path touches prisma/migrations/*.sql while locked, return a { deny } explanation instead of calling next(e). Render a compact prompt-band badge via ui.render showing MIGRATIONS LOCKED in amber or UNLOCKED in emerald. Use pure zero-dependency TypeScript and only the $ Mods API ($.state, $.command, $.ui) with zero npm or node:fs imports. Wrap logic in try/catch so unexpected errors log a warning toast and safely return next(e) without freezing the session.”

Lifecycle of a Session `dev-mod` vs. an Installed Plugin Mod

Stage	Disk Path & Scope	Behavior & Promotion Workflow
1. Live Scratchpad	`~/.claude/dev-mods/<session-id>/<name>/`	Scoped strictly to your current terminal/app session; hot-reloads on every edit so bad experiments never pollute new chats.
2. Personal Daily Mod	`~/.claude/plugins/<name>/`	Validated once with `claude plugin validate`; loads automatically across all your local repositories after `sec-default`.
3. Shared Repo Mod	`<repo>/.claude-plugin/` (committed to Git)	Checked into version control so every teammate gets project-specific guardrails and custom `/commands` after trust approval.

Golden Rule for Vibe-Modding: Always Ask for a Defensive `try / catch`

Because live `dev-mods` hot-reload while you work, instructing Claude to wrap custom logic in `try { ... } catch { return next(e); }` guarantees a typo during rapid iteration degrades gracefully instead of stalling your prompt input.

Customizing the UI: HUDs, Gauges & Custom Panels

Unlike external CLI wrappers, a single `ui.render` handler renders seamlessly in **both** the terminal TUI and the Claude Code Desktop app—using declarative layout primitives (`box`, `row`, `text`, `badge`, `progress`) instead of brittle ANSI escape codes.

A Additive Render Sites

- **`status-line`**: Compact pill metrics pinned beside model and branch indicators.
- **`prompt-band`**: A 1–2 row telemetry strip directly above your typing cursor (ideal for cache TTL & token gauges).
- **`side-pane`**: Docked vertical column for live subagent queues, test checklists, or touched-file summaries.

B Built-In Panel Overrides

- **Custom Diff Viewer:** Intercept the built-in `diff-panel` site to group changes by architectural layer or highlight schema risks.
- **Tool Approval Cards:** Enrich confirmation prompts with blast-radius scores and plain-English explanations before you press Allow.

```
export function register(on: any) {
  // Record timestamp whenever an LLM turn completes so we can track the 5-minute prompt cache window
  on('turn.complete', async ($: any, e: any, next: any) => {
    await $.state.set('lastTurnEpochMs', Date.now());
    await $.state.set('ctxPct', Math.round(((e.contextTokens || 0) / 200000) * 100));
    return next(e);
  });

  // Paint a flicker-free Prompt Cache TTL + Context Gauge directly above the input box
  on('ui.render', { site: 'prompt-band' }, async ($: any, e: any, next: any) => {
    const lastMs = (await $.state.get('lastTurnEpochMs')) || 0;
    const pct = (await $.state.get('ctxPct')) || 0;
    const remainSec = lastMs ? Math.max(0, 300 - Math.floor((Date.now() - lastMs) / 1000)) : 0;
    const cacheLabel = remainSec > 60 ? 'CACHE WARM ' + remainSec + 's' : (remainSec > 0 ? 'COOLING ' + remainSec + 's' :
'CACHE COLD');
    return {
      type: 'row',
      gap: 2,
      children: [
        { type: 'badge', tone: remainSec > 60 ? 'success' : 'warning', text: cacheLabel },
        { type: 'text', text: 'Context Window: ' + pct + '% full' }
      ]
    };
  });
}
```



Write Once, Render Across CLI & Desktop: Declarative `ui.render` trees adapt automatically from compact terminal status bands to crisp split-pane cards inside Claude Code Desktop—keeping cache warmth, context budgets, and subagent progress continuously visible.

5 Real-World Copy-Paste Mod Recipes

Paste any prompt below directly into Claude Code to generate and hot-load the mod inside `~/.claude/dev-mods/` in under a minute—then promote your favorites to `~/.claude/plugins/`.

Mod Recipe	Hooks & `$` APIs Used	Copy-Paste Vibe-Modding Spec / Prompt

1. Prompt Cache TTL & Cost HUD
Cost Saver
	`turn.complete`
	`ui.render`
	`$.state`, `$.timer`
		"Create a zero-npm dev-mod `cache-hud`. On `turn.complete` store timestamp, context %, and cumulative spend in `$.state`. Render a `prompt-band` row via `ui.render` counting down the 300s prompt-cache TTL (emerald >90s, amber <90s, slate when cold) so I send follow-ups before cache expiry. Memoize state per second to prevent flicker."

2. Blast-Radius Shell Interceptor
Safety Shield
	`tool.call` (Bash)
	`command.run`
	`$.ui.toast`
		"Build a zero-npm dev-mod `blast-guard`. Intercept `tool.call` for `Bash`: block `DROP TABLE`, `TRUNCATE`, `git push --force`, and recursive `rm -rf` outside `./tmp` or `./dist` by returning `{ deny }` with a safe alternative. Also register `/blast-radius <cmd>` on `command.run` to score command risk locally with zero LLM tokens."

3. Secret & API-Key Redactor
Zero-Leak Privacy
	`tool.call` (All)
	`prompt.submit`
	`$.state`
		"Create a zero-npm dev-mod `secret-scrub`. Wrap `await next(e)` on `tool.call` (`Bash`, `Read`, `Grep`) and scrub `sk-ant-*`, `sk-proj-*`, `AKIA*`, `ghp_*`, Bearer JWTs, and `.env` assignments from returned output before the model sees them. Keep a redacted counter in `$.state` and display total scrubbed secrets in `status-line`."

4. Rambling Voice-Prompt Structurer
Input Polish
	`prompt.submit`
	`$.model`
	`$.command`
		"Build a zero-npm dev-mod `voice-polish`. When a submitted prompt starts with `v:` on `prompt.submit`, strip filler words and use `$.model.complete` (fast tier) to structure the rambling dictation into Goal, Target Files, and Verification Steps, then pass the clean text via `next({ ...e, prompt })`. Fall back to raw text on timeout."

5. Subagent & Touched-Files Tracker
Swarm Visibility
	`tool.call`
	`subagent.*`
	`ui.render` (side-pane)
		"Create a zero-npm dev-mod `swarm-radar`. Track every file modified by `Write` / `Edit` and active subagent tasks in `$.state`. Render a `side-pane` via `ui.render` listing running subagents at the top and deduplicated modified paths with edit counts underneath, plus a `/reset-radar` local slash command to clear the list."

Stacking All Five Safely: Keep Each Mod Focused on One Job

Because Claude Code composes mods sequentially through `next(e)`, five small 60-line single-purpose mods are dramatically easier to debug, validate, and toggle on or off than one monolithic 500-line mega-mod.

Security, Load Order & `sec-default` Governance

Understanding one architectural truth prevents serious security mistakes: **Claude Code Mods run in-process with your full OS user privileges**—they are capability-audited via `$`, not isolated inside a hypervisor VM.

Treat Third-Party `register.ts` Files Like Local Shell Scripts or Git Hooks

Never clone and approve an untrusted mod without reading its short `register.ts` source and running `claude plugin validate`. Because mods are zero-dependency and typically 40–120 lines of plain TypeScript, a full human + static capability audit takes less than sixty seconds.

Why Slot #1 Belongs to `sec-default`

In middleware queues, **whoever registers first owns the outermost gate**. On Team and Enterprise environments (or wherever managed settings exist), Anthropic's built-in `cc-plugin-sec-default@builtin` mod is pinned to **Slot #1** ahead of every plugin and session `dev-mod`:

- Prevents user mods from reading or stripping managed system prompts and org `CLAUDE.md` policies.
- Enforces admin tool denials and MCP server allowlists before user handlers ever see the event.
- Protects audit telemetry from downstream tampering.

Deterministic 4-Tier Load Order

1. `sec-default` **(Built-In Org Guard):** Immutable Slot #1 baseline when managed policies are active.
2. **Enterprise Managed Mods:** Organization-distributed compliance, redaction, and logging mods.
3. **Installed User & Repo Mods:** Validated plugins in `~/.claude/plugins/` and trusted repo bundles.
4. **Live Session** `dev-mods`**:** Ephemeral scratchpad mods in `~/.claude/dev-mods/<session-id>/`.

Earlier mods inspect inputs first and inspect returned outputs last.

Locking Down Enterprise Workstations via `managed-settings.json`

Security teams can configure `sec-default` to allow only IT-signed managed mods (blocking ad-hoc personal plugins and `dev-mods` on regulated machines) while enforcing capability validation in GitHub Actions CI:

```
// Enterprise managed-settings.json — locks workstations to IT-approved managed mods only:
{
  "pluginConfigs": {
    "cc-plugin-sec-default@builtin": {
      "options": {
        "allowManagedModsOnly": true,
        "protectManagedPrompts": true
      }
    }
  }
}

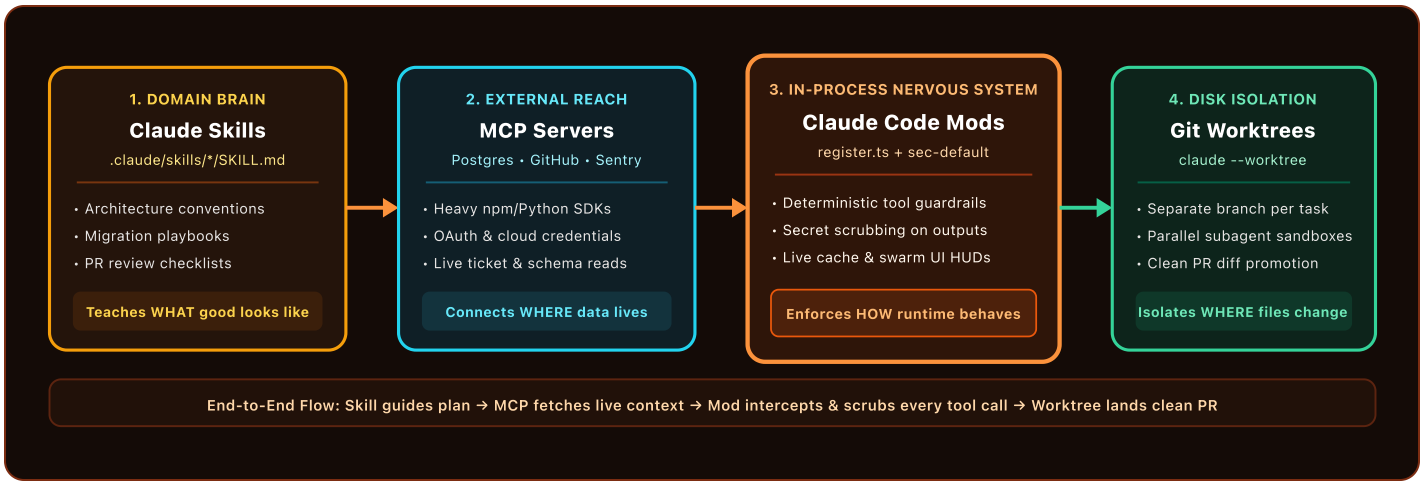
# CI Gate (.github/workflows/validate-mods.yml) — reject PRs if any repo mod fails static scanning:
claude plugin validate ./tools/claude-mods/ops-shield --strict
```

Smooth Team Rollout Tip: Ship New Org Mods in "Warn-Only Toast Mode" First

Before flipping a company-wide managed mod to hard `{ deny }` blocking, run it for one sprint in observe mode—logging matched commands via `$.ui.toast()` and passing events through `next(e)` so platform engineers can tune false positives without interrupting developer flow.

Cross-Tool Synergy: The 4-Layer Agent Stack

Mods do not replace Skills, MCP servers, or Git worktrees—they complete the stack by supplying the missing deterministic runtime layer that keeps autonomous multi-agent coding fast, visible, and safe.



Walkthrough: How All Four Layers Cooperate on a Production Incident

1 Skill + MCP Gather Context

You launch `claude --worktree hotfix-webhook`. Your repository's `incident-triage/SKILL.md` instructs Claude on service boundaries while the Sentry and Postgres **MCP servers** pull the failing stack trace and row metadata.

2 Mods Guard & Instrument Every Turn

As Claude queries staging and edits TypeScript inside the isolated worktree, your **Mods** scrub customer PII from MCP tool outputs before context ingestion, block any write SQL against production, and keep your prompt-cache TTL warm in the bottom HUD.

Why Soft Prompt Rules Alone Are Never Enough for Guardrails

Even the best `CLAUDE.md` or `SKILL.md` rule is probabilistic—under a 180k-token context window, models occasionally forget soft instructions. Moving hard invariants (blocking `--force`, redacting tokens, locking migration directories) into a 30-line **Mod** makes compliance 100% deterministic.

Troubleshooting Clinic & Pre-Flight Checklist

Ninety-five percent of mod hiccups trace back to one of seven predictable causes. Use this table for ten-second fixes before shipping a mod to your daily workflow or team repository.

Symptom / Error	Root Cause	10-Second Fix
1. Capability scanner rejects mod on validate/install	`register.ts` imports an `npm` package, `node:fs`, `process`, or a `.wasm` binary.	Delete external imports; use native ES syntax and `$.fs`, `$.process`, or `$.state`. Move heavy SDK calls into an MCP server.
2. Session freezes right after submitting a prompt or tool	A middleware handler exited a branch without returning `await next(e)` or a terminal response object.	Audit every `if / else` and `catch` path in `register.ts` to ensure it returns either `next(e)` or an explicit `{ deny }` / `{ text }` result.
3. Session `dev-mod` ignored or permission override blocked	`sec-default` in Slot #1 enforces enterprise denials or `allowManagedModsOnly: true`.	Inspect managed policy via `/status`; user mods cannot weaken org denials. Request IT promotion if the mod belongs fleet-wide.
4. Status bar or HUD flickers on every keystroke	`ui.render` runs slow shell subprocesses or recomputes uncached values synchronously on each frame.	Never call `$.process.exec` inside `ui.render`! Compute state asynchronously on `turn.complete` / `tool.call`, store in `$.state`, and read memory in `ui.render`.
5. Module not found for `./register.ts`	Path in `hooks/hooks.json` was written relative to repo root (`./hooks/register.ts`) instead of `hooks/`.	Set `"modules": ["./register.ts"]` inside `hooks/hooks.json` and keep `register.ts` right beside `hooks.json`.
6. `"modules"` key silently ignored at startup	Local Claude Code binary predates the `v2.1.287` Mods engine release.	Run `claude update` (or upgrade Claude Code Desktop) and confirm `claude --version` reports `2.1.287` or higher.
7. Two mods conflict on the same `/command` or rewrite	Both mods short-circuit `command.run` or overwrite `e.prompt` instead of composing cleanly.	Namespace custom slash commands (e.g., `/ops-check`) and mutate `next({ ...e })` incrementally rather than discarding prior fields.

6-Point Pre-Flight Checklist Before Sharing or Committing a Mod

- ☑ **Version Floor Verified:** Confirmed `claude --version` is `v2.1.287+` across CLI and Desktop.
- ☑ **Zero External Dependencies:** Zero `npm`, React, WASM, or raw Node built-ins—only clean `$` API calls.
- ☑ **Every Branch Returns `next(e)`:** Wrapped custom logic in `try/catch` with a safe `return next(e)` fallback.
- ☑ **Pure Memory Reads in `ui.render`:** Shell/model work cached in `$.state` on events; zero I/O inside render frames.
- ☑ **Static Scanner Clean:** Ran `claude plugin validate ./mod-dir` with zero schema or capability warnings.
- ☑ **Source Audited Line-by-Line:** Reviewed full `register.ts` knowing mods run in-process with user privileges.