OPEN HARNESSES SERIES • EXTENSIONS & AUTOMATION GUIDE

OpenCode Plugins, Custom Tools, Servers & CI

Extend OpenCode v2 from an interactive terminal into a programmable engineering platform—registering typed custom tools via `ctx.tool.transform()`, intercepting calls with `@opencode/plugin` hooks, enforcing ordered `permissions` firewalls, running headless `opencode serve` daemons, and automating PRs in GitHub Actions.

Typed `ctx.tool.transform()` | `@opencode/plugin` Hooks | `allow / ask / deny` Policy | Code Mode & Remote MCP | `opencode serve` & SDK | GitHub Actions `run --auto`



In-Process Plugins & Tools

Drop TypeScript plugins using `Plugin.define()` into `.opencode/plugins/` to add domain tools and lifecycle guards instantly.

Deterministic Safety Gates

Combine ordered `permissions` rules in `opencode.json` with `ctx.tool.hook("execute.before")` secret and shell shields.

Headless Daemons & CI

Drive remote sessions over HTTP via `opencode serve`, `opencode --server`, `opencode pair`, `@opencode/client`, and CI runs.

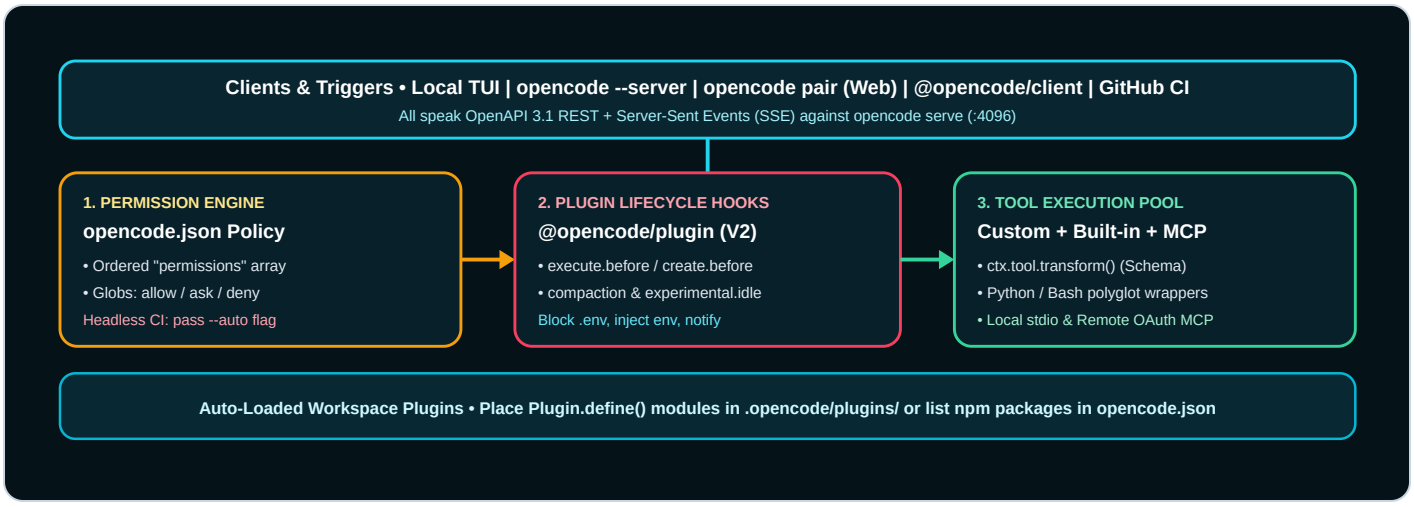
Programmable Inside and Out: Tools, Hooks, MCP & Headless HTTP

Every OpenCode session runs on an HTTP/SSE server engine under the hood. That architecture lets you extend the agent at four clean layers: **custom TypeScript tools**, **event & interceptor plugins**, **external MCP servers**, and **headless CLI / SDK automation**.

Quick Navigation

1. Four-Layer Extension Model p. 2
2. `.opencode/` Layout & Decision Matrix p. 3
3. Annotated Server & Plugin Tour p. 4
4. Authoring Custom Tools (Zod) p. 5
5. Plugins, Hooks & Permission Gates p. 6
6. Troubleshooting & Pre-Flight p. 12
7. MCP Servers & OAuth Bridges p. 7
8. Headless `serve`, `run` & CI p. 8
9. 5 Copy-Paste Production Recipes p. 9
10. Server Hardening & SDK Control p. 10
11. Cross-Tool Synergy Pipeline p. 11

Architecture: How Custom Tools, Lifecycle Hooks & Headless Clients Interlock



Zero-Build Extension Ergonomics

Drop any V2 `Plugin.define()` directory or file into `.opencode/plugins/` (or global `~/.config/opencode/plugins/`). OpenCode loads TypeScript server, TUI, and RPC exports directly and resolves host `@opencode/plugin` and Effect dependencies cleanly at startup.

The `.opencode/` Directory Map & Extension Decision Matrix

Every customization in OpenCode has a dedicated folder in your project root (shared with teammates via Git) and a matching personal folder under `~/.config/opencode/` (shared across all your repos).

1 Standard `.opencode/` Project Layout

```
my-repo/
├── opencode.json              # Models,
permissions[], mcp, plugins[]
├── AGENTS.md                  # Repo architecture &
test rules
└── .opencode/
    ├── package.json           # Shared npm deps for
plugins
    ├── plugins/               # V2 tools, hooks &
TUI/GUI mods
    │   ├── metrics/index.ts   # Custom tools via
ctx.tool.transform
    │   └── guard/index.ts     # execute.before &
shell guards
    ├── agents/                # Specialist Markdown
subagents
    └── commands/              # Reusable /slash
prompt templates
```

2 Loading npm Plugins vs. Local Files

Modules in `.opencode/plugins/` auto-load cleanly via `Plugin.define`. Published or local directories with options go in `opencode.json` under plural `"plugins"`:

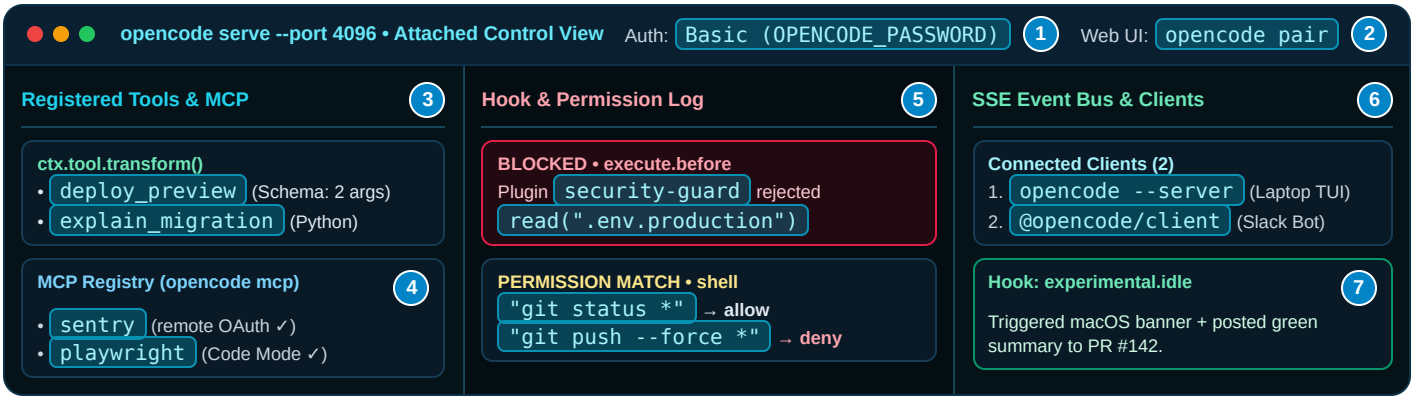
```
{
  "$schema": "https://opencode.ai/config.json",
  "plugins": [
    "opencode-devcontainers",
    { "package": "@my-org/audit-plugin@2.0.0",
"options": { "strict": true } }
  ]
}
```

Extension Mechanism Decision Matrix: Pick the Simplest Primitive

Primitive	Where It Lives	Use It When You Want To…	Avoid Using It For…
Slash Command	`.opencode/commands/*.md`	Trigger a repeatable prompt macro manually (e.g., `/changelog`, `/pr-check`)	Tools the LLM must decide to invoke mid-turn on its own
Custom Tool	`.opencode/plugins/*/index.ts`	Register typed domain actions via `ctx.tool.transform()` (SQL checks, Python wrappers)	Cross-app SaaS integrations that already ship an official MCP server
Plugin Hook	`.opencode/plugins/*/index.ts`	Enforce security blocks (`execute.before`), shell env (`create.before`), or compaction	Simple static prompt rules that belong in `AGENTS.md`
MCP Server	`opencode.json` → `mcp.servers`	Connect multi-tool ecosystems (Sentry, Linear, GitHub, Playwright, Postgres)	Disabling Code Mode on 40+ tools globally (bloats every turn)

Annotated Tour: Inspecting Plugins, Permissions & `opencode serve`

When running OpenCode as a persistent daemon (`opencode serve` paired via `opencode pair`) with remote TUI/HTTP clients, every plugin interception, MCP OAuth status, and permission gate surfaces deterministically.



Control Legend: Seven Critical Automation Surfaces

1 **HTTP Basic Auth Guard:** Setting `OPENCODE_PASSWORD` protects headless `opencode serve` HTTP/web ports (fixed username: `opencode`).

2 **One-Time Browser Pairing:** Run `opencode pair` to mint a single-use 5-minute pairing URL instead of leaving unauthenticated web sessions open.

3 **Custom Tool Transform:** Plugins register typed tools with JSON Schema or Zod/Standard Schema inside `ctx.tool.transform((ed) => ed.add(...))`.

4 **Code Mode & Scoped MCP:** Manage remote OAuth via `opencode mcp auth <name>`; Code Mode keeps MCP tool schemas compact on subagents.

5 **Dual-Layer Security Gate:** Ordered `permissions` globs handle 90% of rules; imperative `ctx.tool.hook("execute.before")` inspects dynamic arguments.

6 **Remote Attach &** 7 **Idle Hook:** Drive one remote devbox via `opencode --server`, browser pairing, or `@opencode/client` and fire `experimental.idle` hooks on completion.

Authoring Custom Tools with `@opencode/plugin` V2

Custom tools turn tribal shell scripts into typed, schema-validated capabilities inside a V2 plugin's `ctx.tool.transform()` editor—supporting JSON Schema, Standard Schema (Zod), cooperative cancellation, and namespacing.

1. Read-Only SQL Tool (`.opencode/plugins/db/index.ts`)

```
import { Plugin } from "@opencode/plugin"

export default Plugin.define({
  id: "db-tools",
  async setup(ctx) {
    await ctx.tool.transform((ed) => {
      ed.add({
        name: "db",
        description: "Run read-only EXPLAIN on
Postgres",
        input: {
          type: "object",
          properties: { sql: { type: "string" } },
          required: ["sql"],
        },
        execute: async ({ sql }, toolCtx) => {
          if (!/^SELECT\b/i.test(sql.trim())) {
            return { content: "Only SELECT queries
allowed." }
          }
          const out = await Bun.$`psql "$STAGING_RO_URL"
-c "EXPLAIN ${sql}"`.text()
          return { content: `Dir:
${ctx.location.directory}\n${out}` }
        },
      })
    })
  },
})
```

2. Polyglot Python Script + Signal & Namespace

```
// .opencode/plugins/analytics/index.ts
import { Plugin } from "@opencode/plugin"
import path from "path"

export default Plugin.define({
  id: "analytics-tools",
  async setup(ctx) {
    await ctx.tool.transform((ed) => {
      ed.namespace({ name: "ops", description: "Ops
scripts" })
      ed.add({
        name: "forecast",
        description: "Score latency CSV via Python",
        input: {
          type: "object",
          properties: { csvPath: { type: "string" } },
          required: ["csvPath"],
        },
        options: { namespace: "ops", codemode: true },
        execute: async ({ csvPath }, toolCtx) => {
          await toolCtx.progress({ status: "scoring" })
          const s = path.join(ctx.location.directory,
"scripts/score.py")
          const p = Bun.spawn(["python3", s, csvPath], {
signal: toolCtx.signal })
          return { content: await new
Response(p.stdout).text() }
        },
      })
    })
  },
})
```

Key Tool Naming, Override & Context Rules

1 Namespaces & Code Mode (`v2.4+`)

Call `ed.namespace(...)` and pass `options: { namespace, codemode: { description, input, output } }` (or `true`) to customize Code Mode schemas.

2 Rich Context & Cache Entries

`setup(ctx)` exposes `ctx.location.directory` and cache-safe `ctx.session.instructions.entry.put()`; `execute` receives `toolCtx.signal`.

3 Transform Built-Ins

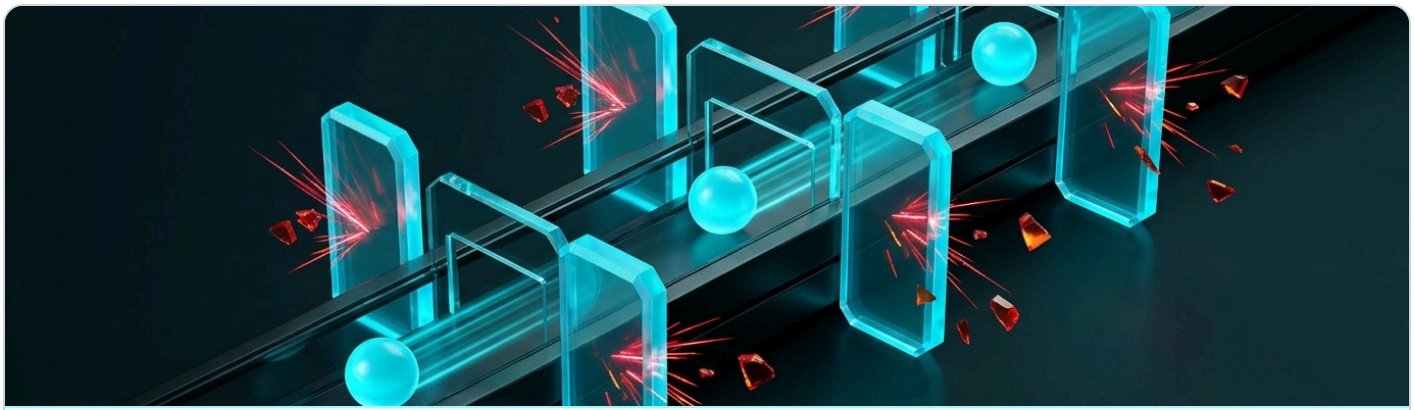
Use `ed.update("bash", ...)` or `ed.remove(...)` inside `ctx.tool.transform()` to wrap or replace OpenCode's built-in tools with custom sandboxed runners.

Keep Tool Returns Compact & Structured

Never return 50,000 raw log lines from `execute()`. Filter, slice, or summarize bulky CLI output inside your TypeScript wrapper and write full dumps to a temp file path if the agent needs deeper drill-down.

Defense in Depth: Plugin Interceptors & `permissions` Firewalls

Prompt instructions alone cannot stop a confused or prompt-injected model from reading `.env` secrets or running destructive shell flags. Combine OpenCode v2's ordered `permissions` array with hard `Plugin.define` runtime hooks.



Deterministic Gatekeeping: Safe read/test commands glide through `"allow"` gates while destructive shell patterns and secret-file reads shatter at the `permissions` and `execute.before` barrier.

1. Ordered `permissions` Array in `opencode.json`

```
{
  "permissions": [
    { "action": "edit",  "resource": "*",
"effect": "allow" },
    { "action": "shell", "resource": "*",
"effect": "ask" },
    { "action": "shell", "resource": "pnpm test *",
"effect": "allow" },
    { "action": "shell", "resource": "git status *",
"effect": "allow" },
    { "action": "shell", "resource": "rm -rf *",
"effect": "deny" },
    { "action": "read",  "resource": "*",
"effect": "allow" },
    { "action": "read",  "resource": "*.env*",
"effect": "deny" }
  ]
}
```

2. Plugin Shield (`.opencode/plugins/guard/index.ts`)

```
import { Plugin } from "@opencode/plugin"

export default Plugin.define({
  id: "security-guard",
  async setup(ctx) {
    await ctx.tool.hook("execute.before", (ev) => {
      if (ev.tool === "read" &&
/\.pem|id_rsa/.test(JSON.stringify(ev.input))) {
        throw new Error("Blocked private key read by
guard")
      }
    })
    await ctx.session.hook("experimental.idle", async ()
=> {
      await Bun.$`osascript -e 'display notification
"Done!"'`
    })
  },
})
```

Color-Coded Blueprint: Anatomy of a Hardened Automation Config

Expose read-only domain tools via ctx.tool.transform(), declare ordered permissions rules allowing pnpm test * and denying *.env* or rm -rf *, enforce execute.before key shields and compaction hooks in .opencode/plugins/guard/, protect opencode serve behind OPENCODE_PASSWORD on Tailscale, and pass --auto (with explicit deny rules) so headless CI runs never stall on ask.

Typed Custom Tools | Declarative Permission | Lifecycle Hooks | Authenticated Daemon | Headless CI Policy

Wiring Local & Remote MCP Servers Without Token Bloat

OpenCode supports both local subprocess MCP servers (`"type": "local"`) and remote HTTP/SSE MCP endpoints with automatic OAuth 2.1 browser flows (`"type": "remote"`)—plus per-agent glob controls so heavy tool schemas only load when needed.

1. Local + Remote MCP under `mcp.servers`

```
{
  "mcp": {
    "servers": {
      "playwright": {
        "type": "local",
        "command": ["npx", "-y",
"@playwright/mcp@latest"]
      },
      "sentry": {
        "type": "remote",
        "url": "https://mcp.sentry.dev/mcp"
      }
    }
  },
  "permissions": [
    { "action": "playwright_*", "resource": "*",
"effect": "deny" }
  ],
  "agents": {
    "qa-browser": {
      "permissions": [{ "action": "playwright_*",
"resource": "*", "effect": "allow" }]
    }
  }
}
```

2. Managing MCP Auth & Health via CLI

```
# Add a remote MCP endpoint interactively or via CLI
opencode mcp add sentry --url https://mcp.sentry.dev/mcp

# Check connection health & status across servers
opencode mcp list

# Complete OAuth browser login for remote servers
opencode mcp auth sentry

# Inspect verbose transport / OAuth logs on failure
opencode --log-level debug --print-logs mcp list
```

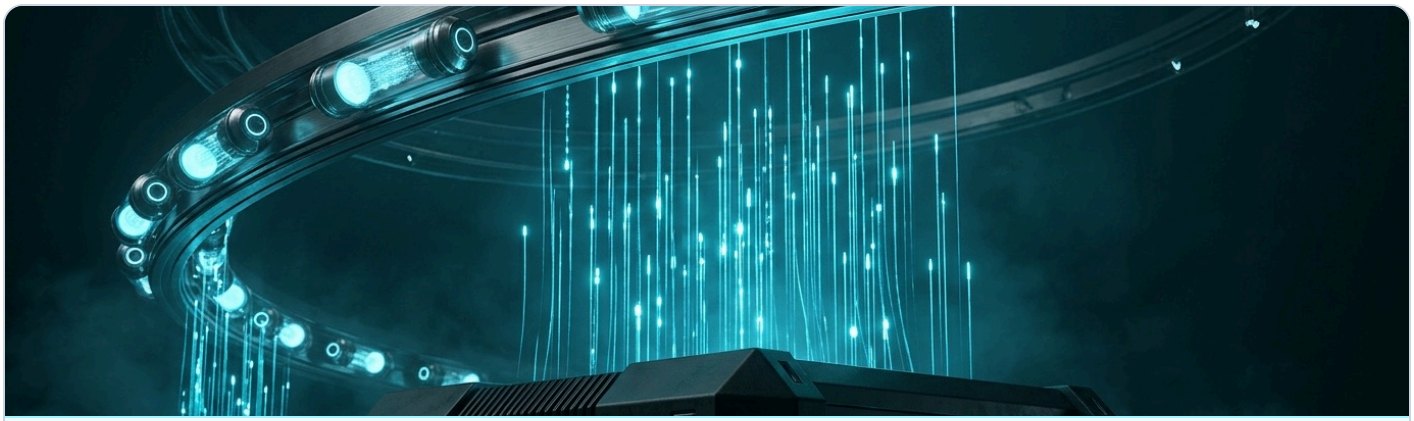
Token & Process Savings: V2 enables `codemode: true` by default, supports **Shared MCP Connection Pooling** across projects (`v2.2.6+`), and scopes `playwright_*` tools per agent!

Custom Compaction Hook Keeps Critical Context Alive

Long MCP debugging sessions eventually trigger compaction. Implement `await ctx.session.hook("compaction", (ev) => ...)` inside a plugin to append must-keep invariants to `ev.system` or supply `ev.result` so compacted summaries retain operational state.

Headless Execution: `opencode serve`, `run` & GitHub Actions

Decouple your local laptop from heavy builds by running `opencode serve` on a cloud devbox, scripting one-shot pipelines via `opencode run --auto`, and triggering autonomous fixes in GitHub Actions.



Always-On Automation Loop: Issue comments, cron triggers, and remote TUI clients dispatch tasks into headless OpenCode workers that run tests in clean containers and push verified commits back to the PR branch.

1. Remote Devbox Daemon + Local Attach

```
# On cloud VM / GPU box (protected over Tailscale)
export OPENCODE_PASSWORD="strong-random-secret"
opencode serve --hostname 0.0.0.0 --port 4096

# From lightweight laptop: attach remote TUI (or run
opencode pair)
opencode --server http://devbox.tailnet:4096

# One-shot non-interactive CLI / shell pipe
git diff | opencode run --auto --format json \
  "Summarize breaking API changes"
```

2. Headless GitHub Actions Bot (`opencode run`)

```
# .github/workflows/opencode.yml
on:
  issue_comment: { types: [created] }
jobs:
  opencode:
    if: contains(github.event.comment.body, '/oc')
    runs-on: ubuntu-latest
    permissions:
      contents: write
      pull-requests: write
    steps:
      - uses: actions/checkout@v4
      - run: npm install -g @opencode/cli
      - env: { ANTHROPIC_API_KEY: ${{
secrets.ANTHROPIC_API_KEY }} }
        run: opencode run --auto --archived --format
json -m anthropic/claude-sonnet-4-6 "Fix failing tests"
```

5 Real-World Copy-Paste Recipes

Drop these five snippets into your repo, CI workflow, or TypeScript service to add custom tools, shell env injection, SDK orchestration, remote TUI attachment, and scheduled repository sweeps.

Recipe & Goal	Copy-Paste Snippet / Command	Expected Outcome
1. Inject Env Vars Into Every Agent Shell	In `Plugin.define` `setup(ctx)`: `await ctx.shell.hook("create.before", (ev) => { ev.env.CI = "1"; ev.env.PAGER = "cat"; })`	Prevents interactive pagers (`less`, Watch-mode Vitest) from hanging agent bash calls.
2. Programmatic Session Control via V2 Client	`import { OpenCode } from "@opencode/client"` `const c = OpenCode.make({ baseUrl: "http://127.0.0.1:4096" })` `const s = await c.session.create({ location: { directory: "/repo" } })`	Lets bots and dashboards drive remote servers (or use `@opencode/sdk` in-process).
3. One-Shot Prompt Against Running Daemon	`opencode run --server http://127.0.0.1:4096 "Run unit tests for billing and report failures"`	Skips MCP/LSP cold-start boot times by reusing the warm `opencode serve` process.
4. Isolated Worktree & Archived CI Run	`opencode run -w fix-tests --auto --archived --format json -m anthropic/claude-sonnet-4-6 "Fix failing test"`	Runs inside a fresh Git worktree (`-w`), hides from inbox (`--archived`), and emits JSON.
5. Pairing, Schedules & Desktop Browser	`OPENCODE_PASSWORD=secret opencode serve --hostname 0.0.0.0 --port 4096` `opencode pair --url https://devbox.tailnet:4096`	Mints a single-use pairing URL, coordinates `experimental.schedules`, and pairs with the `v2.4+` Desktop Browser.

Always Set `CI=1` and `PAGER=cat` for Agent Subshells

Interactive CLI tools like `git log`, `gh`, or `vitest` wait forever for human keypresses unless `CI=1` and `PAGER=cat` are present. Recipe #1's `ctx.shell.hook("create.before")` plugin eliminates hung terminal tools permanently.

Hardening Headless Servers, CI Secrets & Fork Pull Requests

Giving an AI agent shell and write permissions inside an HTTP daemon or CI runner demands strict network, authentication, and fork-isolation boundaries. Lock down these four surfaces before going live.

1 Never Expose Unauthenticated `0.0.0.0`

- **Loopback by Default:** Keep `opencode serve` bound to `127.0.0.1` locally and forward via SSH or Tailscale Serve.
- **Mandatory Basic Auth:** Always export `OPENCODE_PASSWORD` (in V2 the Basic Auth username is always `opencode`; `OPENCODE_SERVER_PASSWORD` remains supported) on LAN/Tailnet sockets.
- **Restrict CORS:** Pass explicit allowed origins via `--cors https://ops.internal` rather than wildcard origins.

2 Defend GitHub Actions Against Prompt Injection

- **Collaborator Gate Only:** Never let untrusted public issue commenters trigger write-scoped CI agents on repos holding cloud secrets.
- **Least-Privilege `GITHUB_TOKEN`:** Scope workflow YAML `permissions:` strictly to `contents: write` and `pull-requests: write`; never pass broad org admin PATs.
- **Pin Action SHAs:** Pin third-party GitHub Actions by commit SHA in regulated repositories.

3 Audit Third-Party Plugins Like Root Code

Plugins listed in `opencode.json` execute inside your host user's runtime process with full filesystem and env access. Pin exact semver versions (never floating `latest`) and vendor critical internal plugins under `.opencode/plugins/`.

4 Containerize Autonomous `opencode run` Loops

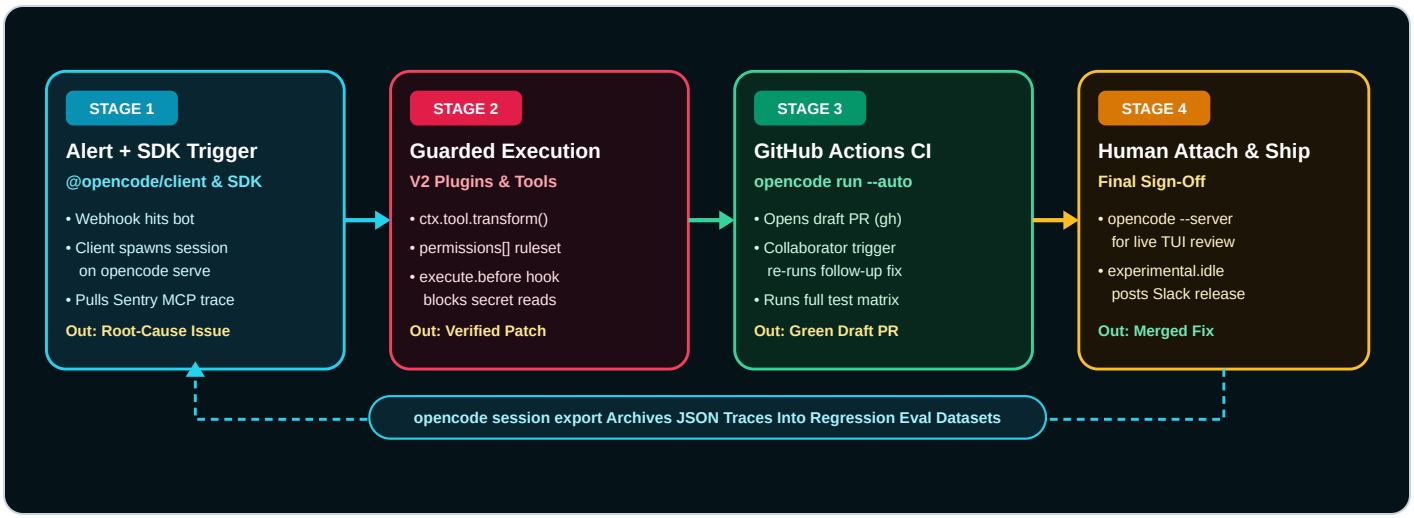
When running broad unattended migrations with `--auto` or `"effect": "allow"` on `shell` (note: V2 allows `external_directory` unless explicitly gated), run inside a rootless devcontainer or ephemeral microVM.

Use V2 Ordered `"permissions"` & `"shell"` Action Names!

In OpenCode v2, permissions use an ordered `"permissions"` array with V2 action names (`"shell"` instead of V1 `"bash"`, `"subagent"` instead of `"task"`). Last matching rule wins—keep `"$schema": "https://opencode.ai/config.json"` at line 1 and never mix V1 and V2 shapes.

Cross-Tool Synergy: Sentry Alert → SDK Triage → Guarded Fix → PR

See how custom tools, MCP connectors, plugin guardrails, headless `opencode serve`, and GitHub Actions unite into an end-to-end incident-to-PR pipeline with human review before merge.



A Warm Daemon Reuse Cuts Script Latency

Starting MCP servers and language servers from scratch on every shell command adds 3–8 seconds of overhead. Keep one background `opencode serve` daemon alive and pass `--server http://127.0.0.1:4096` to every `opencode run` script.

B Shared Team Repo Harness

Checking `.opencode/plugins/`, `.opencode/agents/`, and `opencode.json` into Git gives every new engineer—and your GitHub Actions runner—identical domain tools and safety guardrails on first clone.

Troubleshooting Clinic & 6-Point Pre-Flight Checklist

Resolve common headless CI rejections, Zod version mismatches, hung shell tools, and MCP OAuth errors in seconds, then verify all six release gates before automating repos.

Symptom / Failure	10-Second Root-Cause Fix
1. `opencode run` fails immediately with permission denied	Headless runs have no interactive prompt, so any `"ask"` rule auto-rejects! Pass `--auto` or explicitly set needed rules to `"allow"` for CI.
2. V2 rules in `opencode.json` are ignored or misordered	Use the V2 ordered `"permissions"` array with `"shell"` (last match wins), and check `opencode debug config` for V1/V2 field warnings.
3. V1 plugin throws `PluginModule.LoadError`	V1 function exports no longer load in V2. Export `default Plugin.define({ id, setup })` from `@opencode/plugin` and register tools with `ctx.tool.transform()`.
4. Agent hangs forever running `vitest` or `git diff`	Interactive watch mode or `less` pager is waiting on stdin. Add a `ctx.shell.hook("create.before")` hook setting `CI="1"` and `PAGER="cat"`, and pass `--run`.
5. Remote MCP server returns `401 Unauthorized`	OAuth token expired or wasn't completed. Run `opencode mcp auth <name>` and verify via `opencode --log-level debug --print-logs mcp list`.
6. GitHub Actions CI run silently exits or skips push	Confirm `opencode run --auto` has workflow `contents: write` and `pull-requests: write` permissions and valid provider API secrets.
7. Context tokens spike by 25k before first prompt	Too many MCP servers disabled Code Mode. Keep default `"codemode": true` under `mcp.servers` or deny unused server globs globally.

6-Point Pre-Flight Checklist Before Shipping Plugins & CI

- ✓ **Schema Validated:** Declared `"$schema": "https://opencode.ai/config.json"` and verified ordered `"permissions"` rules.
- ✓ **Dual Secret Shield:** Blocked `*.env*` / `*.pem` in both `permissions` (`read`) and `execute.before`.
- ✓ **Non-Interactive Env:** Injected `CI=1` and `PAGER=cat` via `ctx.shell.hook("create.before")` so shells never hang.
- ✓ **Scoped MCP Schemas:** Used `mcp.servers` with Code Mode enabled and per-agent `permissions` globs.
- ✓ **Authenticated Daemon:** Set `OPENCODE_PASSWORD` and loopback/Tailscale binding for `opencode serve`.
- ✓ **Least-Privilege CI:** Restricted GitHub Actions `run --auto` jobs to collaborators and minimal token scopes.