

# **GSD**

## **The Advanced Guide**

Architecture, security, and organizational deployment

Tibsfox

For GSD v1.35.0

Audience: senior engineers, team leads, architects

# Contents

<b>Preface</b>	<b>5</b>
<b>1 The Orchestrator-Subagent Model</b>	<b>6</b>
1.1 Thin orchestrators, heavy agents . . . . .	6
1.2 Agent lifecycle . . . . .	6
1.3 Why orchestrators don't do heavy lifting . . . . .	7
1.4 State passing . . . . .	7
<b>2 Context Window Management</b>	<b>8</b>
2.1 The context rot problem, quantified . . . . .	8
2.2 Fresh context strategy . . . . .	8
2.3 Adaptive enrichment for 1M-token models . . . . .	9
2.4 Context warnings and the hook layer . . . . .	9
<b>3 XML Prompt Architecture</b>	<b>10</b>
3.1 The <task> format . . . . .	10
3.2 Why XML, not JSON or YAML . . . . .	11
3.3 Task types . . . . .	11
<b>4 Multi-Agent Coordination</b>	<b>12</b>
4.1 The four stages . . . . .	12
4.2 Wave grouping . . . . .	12
4.3 Agent model assignments . . . . .	13
<b>5 Security Model</b>	<b>14</b>
5.1 Threat model . . . . .	14
5.2 Layer 1: Path traversal prevention . . . . .	15
5.3 Layer 2: Prompt injection detection . . . . .	15
5.4 Layer 3: Runtime hooks . . . . .	15
5.5 Layer 4: CI injection scanner . . . . .	16
5.6 Layer 5: Secret deny list . . . . .	16
5.7 A note on --dangerously-skip-permissions . . . . .	17
<b>6 Multi-Runtime Support</b>	<b>18</b>
6.1 Supported runtimes . . . . .	18
6.2 What the installer does per runtime . . . . .	18
6.3 Claude Code 2.1.88+ skills format . . . . .	19

6.4	Non-interactive installation . . . . .	19
6.5	Runtime-specific limitations . . . . .	19
<b>7</b>	<b>SDK and Headless Automation</b>	<b>20</b>
7.1	gsd-sdk . . . . .	20
7.2	Installation and usage . . . . .	20
7.3	CI/CD integration patterns . . . . .	21
<b>8</b>	<b>Git Branching Strategies</b>	<b>22</b>
8.1	Three strategies . . . . .	22
8.2	Templates and naming . . . . .	22
8.3	Squash vs. merge with history . . . . .	22
8.4	<code>/gsd-pr-branch</code> : clean PR branches . . . . .	23
<b>9</b>	<b>Workstreams and Workspaces</b>	<b>24</b>
9.1	Workstreams: parallel planning state . . . . .	24
9.2	Workspaces: full repo isolation . . . . .	24
9.3	When to use which . . . . .	24
9.4	Execution worktree isolation . . . . .	25
<b>10</b>	<b>Nyquist Validation Layer</b>	<b>26</b>
10.1	The feedback contract . . . . .	26
10.2	VALIDATION.md . . . . .	26
10.3	Retroactive validation . . . . .	26
10.4	Toggle . . . . .	27
<b>11</b>	<b>Extension Patterns</b>	<b>28</b>
11.1	Agent skill injection . . . . .	28
11.2	Custom hooks . . . . .	28
11.3	Workflow guard as a template . . . . .	28
11.4	Custom GSD commands . . . . .	29
<b>12</b>	<b>Performance Tuning</b>	<b>30</b>
12.1	Model profile optimization . . . . .	30
12.2	Token budget analysis . . . . .	30
12.3	Parallelization . . . . .	30
12.4	Worktree isolation . . . . .	31
12.5	Context window utilization . . . . .	31
<b>13</b>	<b>Team Adoption Guide</b>	<b>32</b>
13.1	Onboarding playbook . . . . .	32
13.2	Standardizing settings . . . . .	32
13.3	Granularity tuning for team size . . . . .	32
13.4	Code review integration . . . . .	33
13.5	Documentation generation . . . . .	33
13.6	<code>/gsd-milestone-summary</code> for onboarding . . . . .	33
13.7	<code>/gsd-profile-user</code> for personalized workflow adaptation . . . . .	33
<b>A</b>	<b>Security Reference</b>	<b>34</b>

---

A.1 Deny list patterns . . . . .	34
A.2 Runtime hook configuration . . . . .	35
A.3 CI scanner integration . . . . .	35
<b>B Configuration Reference</b>	<b>36</b>
B.1 Top-level structure . . . . .	36
B.2 Convention: absent = enabled . . . . .	38

# Preface

This guide is written for senior engineers, team leads, and architects evaluating GSD for organizational adoption. It assumes a working mental model of distributed systems, agent-based orchestration, CI/CD pipelines, and software supply-chain security. You will not find screenshots or marketing language here. What you will find is a concrete account of how GSD works internally, why it is built the way it is, and what to consider before deploying it across a team.

GSD is a meta-prompting framework. It sits between a human operator and an AI coding runtime (Claude Code, OpenCode, Gemini CLI, Codex, and ten others — see Chapter 6), and converts loose natural-language intent into structured specifications, research, plans, and verified execution. It does not run its own model. It does not ship with any model weights. It ships with a hierarchy of prompts, a CLI of domain modules, a hook system, and an opinion about how to use context windows correctly.

The single observation that drives everything downstream is this: large language models degrade as their context windows fill. GSD calls this *context rot*, and every architectural decision in the system exists to counter it. Once you internalize that premise, the rest of the design falls out naturally.

*GSD is MIT-licensed and developed in the open at [github.com/gsd-build/get-shit-done](https://github.com/gsd-build/get-shit-done). This guide is CC BY-SA 4.0. Both are community artifacts; neither is a commercial product.*

# Chapter 1

## The Orchestrator-Subagent Model

### 1.1 Thin orchestrators, heavy agents

GSD's execution unit is not a single AI session. It is a **thin orchestrator** that spawns **specialized subagents**, each of which runs in its own fresh context window. The orchestrator is a workflow file — a markdown document with no code — that the host runtime (Claude Code, for example) interprets as instructions. It loads a minimal amount of context via the `gsd-tools.cjs` CLI, resolves which agent to spawn, passes that agent a focused prompt payload, waits for the result, and then updates state. The orchestrator itself never writes production code, never reads large source trees, and never reasons at length about the problem. It delegates.

The delegated agents *do* reason at length. Each one is defined by a markdown file under `agents/` with YAML frontmatter naming the agent, describing its role, and declaring its tool access (Read, Write, Edit, Bash, Grep, Glob, WebSearch). When the orchestrator invokes an agent, the runtime spins up a new session, seeds it with the agent's prompt and the orchestrator's context payload, and runs it to completion. The agent returns; the session is discarded.

This asymmetry — thin orchestrator, heavy subagent — is the central design pattern. It exists because reasoning degrades with context length, and orchestration degrades much more slowly than reasoning. By keeping the orchestrator's context near empty and only filling subagent context windows on demand, GSD preserves reasoning quality across an arbitrarily long project.

### 1.2 Agent lifecycle

Each subagent lives through five phases: **spawn**, **context load**, **execute**, **commit**, and **terminate**.

1. **Spawn** — The orchestrator issues a Task/SubAgent call with a model assignment (resolved via `/gsd-tools.cjs resolve-model`), a tool whitelist, and an initial prompt composed of the agent definition and a task payload.

2. **Context load** — The first thing an executor does is read its assigned `PLAN.md`, the phase `CONTEXT.md`, and the phase `RESEARCH.md`. These files define what the agent is supposed to do and why.
3. **Execute** — The agent writes code, runs tests, and records findings. For planners, this means emitting `PLAN.md` files. For executors, it means editing source. For verifiers, it means writing `VERIFICATION.md`.
4. **Commit** — The agent produces one atomic git commit per task, using a conventional-commit message. During parallel waves, commits use `--no-verify` to skip pre-commit hooks; the orchestrator re-runs hooks once at wave end. This avoids build-lock contention (Cargo, npm, Gradle) when multiple agents commit simultaneously.
5. **Terminate** — The session ends. Its context window is discarded. Nothing persists except the files on disk and the commits in git.

### 1.3 Why orchestrators don't do heavy lifting

If you have used a long-running Claude session, you have seen its output quality fall off as the conversation grows. The failure mode is not dramatic. The model does not start producing garbage. Instead, it starts forgetting earlier decisions, missing constraints from thousands of tokens ago, and drifting into plausible-but-wrong answers. This is context rot.

GSD treats context rot as a hard engineering constraint, not a quirk. Orchestrators do not debate design, write code, or synthesize research. They coordinate. Everything that requires actual reasoning — research, planning, coding, verification — happens inside a subagent that was spawned seconds earlier and will be terminated seconds later. The subagent sees exactly the files it needs, reasons over a fresh 200K-token budget (or up to 1M for models that support it), produces its artifact, and exits before context rot has time to set in.

### 1.4 State passing

Because subagents are short-lived and amnesiac, state must live on disk. GSD stores all of it under `.planning/`, a directory tree that contains human-readable Markdown and JSON. The canonical state files are `PROJECT.md` (vision and invariants), `REQUIREMENTS.md` (scope), `ROADMAP.md` (phase plan), `STATE.md` (position, decisions, blockers, metrics), and `config.json` (workflow configuration).

`STATE.md` deserves special attention. It is the living memory of the project. It tracks which phase is active, which plan is being executed, what the executor's last finding was, which blockers exist, and which decisions have been made. Every workflow in GSD reads `STATE.md` at start and updates it at end. Because executors run in parallel, writes are protected by an `O_EXCL`-based lockfile (`STATE.md.lock`) with a 10-second stale-lock timeout and jittered spin-wait. This is the only mutex in the entire system.

## Chapter 2

# Context Window Management

### 2.1 The context rot problem, quantified

Context rot is not a theoretical concern. In practice, Claude (and every other frontier model) starts drifting once a session exceeds roughly 80–120K tokens of accumulated conversation. The drift is continuous rather than discrete: the model does not crash at a specific token, but its adherence to system prompts and earlier constraints degrades measurably. By 160K tokens of accumulated work, complex instructions are often partially forgotten. By 190K, hallucinated paths and missing imports become common.

GSD’s response is not to push against this wall. It is to never approach it. Every subagent is spawned against a **fresh 200K-token window**. The orchestrator itself consumes a small, bounded amount of context — enough for the workflow markdown, the STATE.md snapshot, and the current step. Everything else is handed off.

### 2.2 Fresh context strategy

When an executor agent runs, its context is assembled from exactly four inputs: the agent definition (*agents/gsd-executor.md*), the PLAN.md for the task, the phase CONTEXT.md and RESEARCH.md, and the project-level PROJECT.md and STATE.md. Nothing else. No conversation history. No earlier wave’s work (unless a 1M-token model allows adaptive enrichment; see below). The agent begins its reasoning on an empty slate that contains only the files most likely to be relevant.

This is a deliberate inversion of the usual “load everything into the session and let the model figure it out” approach. Loading everything is cheaper to implement but produces worse results once context passes the rot threshold. GSD pays the coordination cost of agent spawning so that each reasoning step happens against a small, focused payload.

## 2.3 Adaptive enrichment for 1M-token models

For models that support a million-token context window (Claude Opus 4.6, Sonnet 4.6), the orchestrator reads `context_window` from `config.json`. When the value is  $\geq 500,000$ , subagent prompts are automatically enriched. Executors receive prior-wave SUMMARY.md files alongside their own PLAN. Verifiers receive the full set of phase PLAN, SUMMARY, CONTEXT, and RESEARCH files, plus REQUIREMENTS.md. This is opt-in through configuration and gives 1M-class models cross-plan awareness within a phase without risking context rot at smaller window sizes.

For standard 200K windows, prompts use truncated variants with cache-friendly ordering: stable headers first, variable content last. This maximizes the host runtime's prompt-caching hit rate, which in practice halves the time-to-first-token for subsequent agent calls within the same session.

## 2.4 Context warnings and the hook layer

GSD ships a runtime hook, `gsd-context-monitor.js`, that injects advisory warnings into the orchestrator session as its context fills. The hook runs on the runtime's PostToolUse event (Gemini's AfterTool for Gemini CLI) and reads a bridge file written by the statusline hook to estimate remaining tokens. At  $\leq 35\%$  remaining it injects a WARNING ("*avoid starting new complex work*"); at  $\leq 25\%$  it escalates to CRITICAL ("*context nearly exhausted, inform user*"). The hook is advisory-only: it never issues imperative overrides. Debounce is five tool uses between repeated warnings, with severity escalation bypassing the debounce.

## Chapter 3

# XML Prompt Architecture

### 3.1 The <task> format

GSD plans are not natural language. They are structured XML documents emitted by the planner agent and consumed by executor agents. Each plan is a sequence of <task> elements, each containing some subset of the following children: <name>, <files>, <action>, <verify>, <done>. The planner produces them; the executor reads them; the plan-checker validates them before execution begins.

The shape is deliberate and, at first glance, surprisingly rigid. A typical task reads:

```
<task id="03" type="auto">
  <name>Wire STATE.md lockfile into writeStateMd()</name>
  <files>
    src/state/state-io.ts
    src/state/lockfile.ts
    tests/state-io.test.ts
  </files>
  <action>
    Acquire STATE.md.lock via O_EXCL atomic create before any
    writeStateMd() call. Release after fsync. On stale lock
    (mtime > 10s), break and reclaim with jittered spin-wait.
  </action>
  <verify>
    npm test -- tests/state-io.test.ts must pass with zero
    skipped cases. New test covers concurrent writers.
  </verify>
  <done>
    STATE.md writes are serialized under the lockfile with no
    race window between check and write.
  </done>
</task>
```

## 3.2 Why XML, not JSON or YAML

Three reasons, in priority order. First, tag-delimited content survives natural-language drift. An LLM will, with high reliability, respect `<action>...</action>` boundaries even when the content inside contains code samples, angle brackets, or partially-quoted shell commands. JSON requires escaping. YAML depends on indentation the model must emit exactly. XML tolerates messy internal content and is the format frontier models have been trained to respect in Anthropic-style prompts.

Second, XML tags are unambiguous to both the planner and the executor. The planner knows it must emit `<verify>` or the plan-checker will reject the plan. The executor knows that whatever is in `<verify>` is a command it must run before considering the task done. The machine-readable contract between agents is made legible to both by the same syntactic layer.

Third, tasks nest naturally. A plan is a sequence of `<task>` elements. A phase is a directory of plans. A project is a roadmap of phases. XML scales up hierarchically without ambiguity.

## 3.3 Task types

The `type` attribute on a task declares its execution mode. **auto** tasks are executed by the executor agent with no human gate. **manual** tasks pause execution and require the user to complete them (common for UI work, credentials setup, or external service configuration). **test** tasks are verification gates: they must pass but cannot edit source. The plan-checker validates that every task has a sound `<verify>` block; tasks with unverifiable `<done>` conditions are rejected.

## Chapter 4

# Multi-Agent Coordination

### 4.1 The four stages

A GSD phase runs through four stages, each producing specific artifacts:

1. **Research** — 4 parallel researcher agents probe stack, features, architecture, and pitfalls, writing STACK.md, FEATURES.md, ARCHITECTURE.md, and PITFALLS.md. A separate synthesizer agent then reads all four and produces SUMMARY.md.
2. **Planning** — The planner agent reads SUMMARY.md, REQUIREMENTS.md, and the phase CONTEXT.md, and emits one or more PLAN.md files in XML. The plan-checker then validates the plans for verifiable <done> conditions, wave independence, reachability, and the eight verification dimensions (see Chapter 10 for Nyquist). If the checker rejects, the planner revises. This loop runs up to three iterations; after three, the workflow halts for human review.
3. **Execution** — The orchestrator performs wave analysis, grouping plans by dependency. Independent plans form Wave 1, their dependents form Wave 2, and so on. Within a wave, executors run in parallel — one per plan. Each executor produces code, atomic commits, and a SUMMARY.md.
4. **Verification** — A verifier agent reads all phase SUMMARY.md files, the REQUIREMENTS.md, and the phase goals. It runs the configured test suite, checks each requirement against the delivered work, and writes VERIFICATION.md. If gaps exist, the planner is re-invoked in gap-closure mode (*references/planner-gap-closure.md*) to emit a targeted fix plan.

### 4.2 Wave grouping

Wave grouping is computed from the <depends> declarations inside each task (or, for cross-plan dependencies, from the plan-level <requires> list). The algorithm is a topological sort that partitions plans into the minimum number of levels such that every plan's dependencies are in earlier levels. Within a level, plans are independent and run in parallel. Across levels, execution is strictly sequential.

This matters because the *observed* performance gain from parallelism is large: a phase with four independent plans executes roughly four times faster than it would sequentially, bounded by the slowest plan in the wave. For CI-class workloads where each plan takes 5–15 minutes, the difference between 60-minute and 15-minute phase turnaround is the difference between watching a build and getting on with your day.

### 4.3 Agent model assignments

GSD assigns models per role, resolved via the `/gsd-tools.cjs resolve-model` command against the configured profile. A typical balanced profile looks like this:

Role	Typical model	Rationale
Project researcher	Sonnet	Broad reasoning, budget-sensitive
Phase researcher	Sonnet	Same as above
Planner	Opus	Highest-quality structured output
Plan-checker	Opus	Adversarial review of planner
Executor	Sonnet	High throughput, cost-efficient
Verifier	Opus	Final gate, quality matters
Debugger	Opus	Diagnostic reasoning
UI auditor	Opus	Visual judgment

Profiles are swappable through `/gsd-set-profile`. The `max` profile assigns Opus everywhere at the cost of significant token spend. The `budget` profile uses Haiku for research and Sonnet for planning. The `inherit` profile defers all selection to the runtime, which matters for non-Anthropic backends like OpenRouter, OpenCode’s `/model` selector, or locally-hosted models.

# Chapter 5

## Security Model

This chapter documents GSD's defenses. It does not document attack techniques. The threat model below exists to explain *why* each defense layer is present; no section here will show you how to craft an injection payload, bypass a guard, or extract secrets. That asymmetry is deliberate.

### 5.1 Threat model

GSD processes markdown files as inputs to large language models. A markdown file on disk is, from the model's perspective, a system-level prompt: whatever it contains becomes part of the agent's context and is interpreted with the same trust as the user's instructions. This creates an attack surface that traditional software does not have. If an attacker can cause a hostile markdown file to land in your project — through a pull request, a scraped dependency, a compromised template, or a typosquatted package — that file can issue instructions that your AI coding agent will cheerfully execute.

The threats GSD takes seriously are:

- **Indirect prompt injection** — hostile content in a project file attempting to override agent instructions.
- **Path traversal** — a supplied file path escaping the project directory to read or write files elsewhere on the host.
- **Secret exfiltration** — agent-driven reads of `.env`, credentials files, or private keys as incidental collateral of a legitimate task.
- **Unscoped edits** — agent-driven writes to files the current workflow did not declare.

Against all four, the system takes a defense-in-depth posture: multiple independent layers, each sufficient to catch a subset of attacks, none trusted as the sole line of defense.

### Defense in depth

The five layers below are independent. A bypass of one does not bypass the others. None of them is a silver bullet; none claims to be. Their value is in their combination.

## 5.2 Layer 1: Path traversal prevention

All file paths supplied to GSD CLI commands pass through the path validator in *get-shit-done/bin/lib/security.cjs*. The validator normalizes the path, resolves symlinks, and asserts that the final location is contained within the project directory. Any attempt to escape via `../` sequences, absolute paths, or symlinked detours throws before the path is used. On macOS, the symlink handling is explicit: the validator follows symlinks all the way to the real path before comparing against the project root, because macOS caches symlinked `/var` and `/tmp` prefixes that naively-compared paths will pass.

This layer defends the CLI tools — state updates, template fills, phase operations — from path-based escape. It does not defend against the agent itself reading outside the project, which is the job of layer 5.

## 5.3 Layer 2: Prompt injection detection

The *security.cjs* module exposes a `scanForInjection(text)` function that the runtime hooks use to inspect content written to `.planning/` files. The function applies two pattern families. The first is a list of instruction-mimicking patterns: content that structurally resembles a system prompt, a role-override directive, or an attempt to introduce fabricated tool-call syntax. The second is an encoding-obfuscation family that catches content attempting to hide instructions inside base64, hex, or unicode-escape disguises. Each pattern carries a finding code and a human-readable message; when the scanner matches, the hook logs the finding with file, pattern, and line.

The scanner is **advisory**, not blocking. This is intentional. False positives are preferable to false negatives in a field where new attack techniques appear monthly, and the scanner is tuned to err on the side of flagging. A blocking scanner would create a usability crater; an advisory one surfaces suspicious content without preventing legitimate work.

## 5.4 Layer 3: Runtime hooks

Two hooks register against the runtime's `PreToolUse` event (`AfterTool` on Gemini CLI):

- *gsd-prompt-guard.js* triggers on Write/Edit operations targeting `.planning/` files. It runs a subset of the *security.cjs* injection patterns inline (the hook is deliber-

ately self-contained so it can ship independent of the CLI package). Detections are logged; the operation proceeds.

- `gsd-workflow-guard.js` triggers on Write/Edit operations targeting files *outside* `.planning/`. It detects edits that happen with no active `/gsd-*` command or Task subagent in context — that is, edits outside any GSD workflow. Detections are advisory: the hook recommends `/gsd-quick` or `/gsd-fast` for state-tracked changes. This hook is **opt-in** via `hooks.workflow_guard: true` in `config.json`; its default is false because it produces noise in mixed-use repos.

Both hooks wrap in try/catch and exit silently on any internal error, a conservative posture that prevents a malformed hook from blocking legitimate work.

## 5.5 Layer 4: CI injection scanner

The repository ships a CI-side scanner at `tests/prompt-injection-scan.test.cjs`, wrapped by the shell driver `scripts/prompt-injection-scan.sh`. This scanner runs the full `security.cjs` pattern library against every markdown file in the repository and fails the test suite if any high-severity pattern matches in an unexpected location. It is the gate that prevents a hostile upstream commit — from a pull request, a dependency update, or a rogue contributor — from landing in `main` without being flagged.

Teams that integrate GSD into shared repositories should run this test in CI. It catches the class of attacks that the runtime hooks (layer 3) would catch at edit time, but does so in a centralized, reviewable place, and produces an auditable paper trail when it fires.

## 5.6 Layer 5: Secret deny list

The final layer is not a GSD component. It is the host runtime's own file-access deny list. Claude Code, OpenCode, and most other runtimes support a per-project `settings.json` with a `permissions.deny` array. GSD recommends adding, at minimum:

```
{
  "permissions": {
    "deny": [
      "Read(.env)",
      "Read(.env.*)",
      "Read(**/credentials.json)",
      "Read(**/*.pem)",
      "Read(**/*.key)",
      "Read(**/id_rsa)",
      "Read(**/id_ed25519)",
      "Read(**/.aws/**)",
      "Read(**/.ssh/**)",
      "Read(**/secrets/**)"
    ]
  }
}
```

```
| }
```

This list prevents the agent — and therefore any subagent it spawns — from reading these files at all, regardless of which GSD command is active. It is enforced by the runtime, not by GSD. It is the last line and the strongest: even if every other layer is defeated, the runtime simply refuses the read.

## 5.7 A note on `--dangerously-skip-permissions`

GSD is designed for frictionless automation, and the upstream README recommends running Claude Code with `--dangerously-skip-permissions` to avoid constant interactive approval prompts. This flag is exactly what it says: it disables the runtime's permission checks for tool calls, so the agent can run arbitrary commands without per-operation consent. It is not inherently unsafe when combined with the layers above, but it removes one layer of friction that would otherwise catch user mistakes.

### Granular alternative

If `--dangerously-skip-permissions` is unacceptable for your team's threat model — and it legitimately is for many — configure granular per-tool permissions in the project `.claude/settings.json` instead. The relevant keys are `permissions.allow` (an allow-list of tool patterns that are approved without prompting) and `permissions.deny` (the hard block list from layer 5). A typical allow list reads `["Read(**)", "Edit(**)", "Write(.planning/**)", "Bash(npm test:*)", "Bash(git status:*)"]` — expanded carefully so the agent can operate but cannot surprise you. This configuration gives you most of the automation benefit of the flag without giving up the runtime permission layer entirely.

## Chapter 6

# Multi-Runtime Support

### 6.1 Supported runtimes

As of v1.35.0, GSD supports fifteen AI coding runtimes. The installer detects or accepts a flag for each and translates GSD’s Claude-Code-native artifacts into the appropriate format at install time. The supported list:

Runtime	Flag	Global location	Command form
Claude Code	--claude	<i>/.claude/</i>	<i>/gsd-command</i> (slash)
OpenCode	--opencode	<i>/.config/opencode/</i>	<i>/gsd-command</i> (slash)
Gemini CLI	--gemini	<i>/.gemini/</i>	<i>/gsd-command</i> (slash)
Kilo	--kilo	<i>/.config/kilo/</i>	<i>/gsd-command</i> (slash)
Codex	--codex	<i>/.codex/</i>	<i>\$gsd-command</i> (skill)
Copilot	--copilot	<i>/.github/</i>	<i>/gsd-command</i> (slash)
Cursor CLI	--cursor	<i>/.cursor/</i>	<i>/gsd-command</i> (slash)
Windsurf	--windsurf	<i>/.codeium/windsurf/</i>	<i>/gsd-command</i> (slash)
Antigravity	--antigravity	<i>/.gemini/antigravity/</i>	skill
Augment Code	--augment	<i>/.augment/</i>	skill
Trae	--trae	<i>/.trae/</i>	skill
Qwen Code	--qwen	<i>/.qwen/</i>	skill
CodeBuddy	--codebuddy	<i>/.codebuddy/</i>	skill
Cline	--cline	<i>.clinerules</i>	rules file
All / interactive	--all	multi-select	varies

Every runtime can also be installed locally to the current project (substitute `--local` for `--global` on any of the flags above; Claude Code becomes *./.claude/*, OpenCode becomes *./.opencode/*, and so on).

### 6.2 What the installer does per runtime

The installer (*bin/install.js*, roughly 3,000 lines) performs runtime-specific translation. Claude Code is used as-is. OpenCode and Kilo share a conversion pipeline that produces flat commands plus subagent-mode agent files. Codex generates a TOML configuration plus skill entries rather than slash commands. Copilot maps tool names — Claude’s Bash becomes *execute*, Read becomes *read*. Gemini CLI adjusts hook event names so *PostToolUse* becomes *AfterTool*. Antigravity, Trae,

Qwen Code, Augment, and CodeBuddy are all skills-first runtimes; the installer converts commands into *SKILL.md* files and adjusts agent references accordingly. Cline is the odd one: it installs a *.clinerules* file that encodes the GSD workflow as rule-based guidance rather than as slash commands.

### 6.3 Claude Code 2.1.88+ skills format

Claude Code 2.1.88 and later use a skills format instead of loose command files: each GSD command becomes a directory under *skills/gsd-\*/SKILL.md*. The installer auto-detects the runtime version and writes the appropriate format. Older Claude Code versions use *commands/gsd/*; newer versions use skills. Downgrading a runtime after install requires a reinstall.

### 6.4 Non-interactive installation

For Docker images and CI environments, pass the runtime flag plus `--global` or `--local` and the installer runs without prompts. The `--uninstall` flag removes all GSD files, hook registrations, and settings entries; a *gsd-file-manifest.json* is written at install time so uninstall is exhaustive. Since v1.17, locally-modified files are backed up to *gsd-local-patches/* for later reapplication via `/gsd-reapply-patches`.

### 6.5 Runtime-specific limitations

Not every runtime supports every feature. Gemini CLI uses AfterTool hooks and does not wire up the context-monitor bridge file identically to Claude Code. Codex's skill model has a 50,000-character per-skill limit that forced the decomposition of *agents/gsd-planner.md* into a core file plus reference modules. Copilot's tool mapping does not support every Claude tool name; workflows that depend on exotic tools degrade gracefully. Cline's rules-based integration does not support subagent spawning; GSD on Cline runs as a single-agent guided workflow rather than a multi-agent orchestration. If multi-agent work is essential to your adoption plan, prefer Claude Code, OpenCode, Kilo, Gemini CLI, or Cursor.

# Chapter 7

## SDK and Headless Automation

### 7.1 gsd-sdk

Beyond the interactive workflows, GSD ships an SDK package, `gsd-sdk`, that exposes headless, programmatic project initialization and execution. The SDK is designed for CI/CD pipelines, bulk project seeding, and automated scenario testing. It wraps the same `gsd-tools.cjs` domain modules that the interactive workflows use, exposing them as typed function calls rather than shell commands.

### 7.2 Installation and usage

The SDK installs via `npm install gsd-sdk` and exposes a small surface: functions for initializing a new project, reading and patching `STATE.md`, invoking agents programmatically, and querying phase progress. A typical headless init looks like:

```
import { initProject, addPhase, runPhase } from 'gsd-sdk';

await initProject({
  name: 'billing-service',
  description: '...',
  runtime: 'claude',
  profile: 'balanced',
  granularity: 'medium'
});

await addPhase({ name: 'schema-migration', goals: [...] });
await runPhase({ phase: '01', autonomous: true });
```

The `autonomous: true` flag disables interactive gates and runs the full plan-execute-verify loop without human input. Any failure at any stage produces a structured error that the caller can branch on.

### 7.3 CI/CD integration patterns

The intended CI pattern is: a nightly job that reads a queue of feature requests, runs `initProject` or `addPhase` for each, executes them headlessly, and opens pull requests against the target repository. GSD's atomic commits make each task independently revertible, so failed phases can be rolled back cleanly. The CI job runs under a service account whose runtime is configured with the deny list from Chapter 5 layer 5, plus a narrow allow list that constrains the job's reach to the target repository.

Headless mode is incompatible with manual-type tasks. Plans that contain manual steps — credential provisioning, third-party integration, UI signoff — will pause and wait for input that never arrives. The SDK exposes a `rejectManual: true` flag that causes the planner to refuse to emit manual tasks, producing an error at plan time instead of a hang at execute time.

## Chapter 8

# Git Branching Strategies

### 8.1 Three strategies

GSD supports three git branching strategies, selected via `workflow.branching` in `config.json`:

- **none** (default) — Everything commits to the current branch. Simplest model, fastest iteration, but requires discipline on shared branches.
- **phase** — One branch per phase, named from a template (default: `gsd/phase-{number}-{slug}`). Merges back to the parent branch on phase completion via `/gsd-complete-phase` or `/gsd-ship`.
- **milestone** — One branch per milestone, with all phases in the milestone committing to that branch. Merges back on `/gsd-complete-milestone`.

The branching decision is orthogonal to the commit granularity decision. You can run phase branching with fine-grained atomic commits inside the branch, or you can run no-branching with squash merges on phase completion. The strategies compose.

### 8.2 Templates and naming

Branch names are generated from templates with variable substitution. The available variables are `{number}` (phase or milestone number), `{slug}` (url-safe phase name), `{date}` (ISO date), and `{user}` (current git user). Configure via `workflow.branch_template` in `config.json`. A common team convention is `feature/{user}/{slug}-phase-{number}`, which embeds ownership and context in the branch name directly.

### 8.3 Squash vs. merge with history

Squash merges collapse the entire phase into one commit, which loses the per-task bisect granularity GSD provides. Merge commits with full history preserve that

granularity at the cost of a busier main. GSD's recommendation is: **merge with history for phases shorter than ten commits; squash for longer phases where the per-task detail is no longer useful**. Configure via `workflow.merge_strategy`.

## 8.4 `/gsd-pr-branch`: clean PR branches

The `/gsd-pr-branch` command creates a clean branch filtered of `.planning/` commits. The original branch keeps its full history; the PR branch contains only the source-code commits. This is the recommended way to open pull requests from a GSD-active branch: reviewers see clean, on-topic commits without being buried in planning churn. Under the hood, the command uses `git filter-branch` or (in newer git) `git filter-repo` to strip the unwanted paths, then force-pushes the filtered branch to the remote under a `-pr` suffix.

## Chapter 9

# Workstreams and Workspaces

### 9.1 Workstreams: parallel planning state

A **workstream** is a parallel planning state inside a single repository. Workstreams isolate `.planning/` state so that two milestones — say, a performance overhaul and a security audit — can be planned and executed in parallel without their ROADMAPs, PLANs, and STATE files colliding. Each workstream has its own `.planning/` subtree; the active workstream is tracked via a session-scoped pointer that defaults to the most-recently-used.

The workstream lifecycle is managed via `/gsd-workstreams`: `create`, `switch`, `list`, `complete`. Creating a new workstream forks the current one's `PROJECT.md` but starts fresh REQUIREMENTS, ROADMAP, and STATE. Switching repoints the active pointer so subsequent commands read and write the switched-to state. Completing a workstream archives it to `.planning/workstreams/archive/`.

### 9.2 Workspaces: full repo isolation

A **workspace** is a full repository copy — either a git worktree or a clone — that lets multiple agents work on multiple phases without stepping on each other's working tree. Workspaces are heavier than workstreams: they duplicate the source tree, consume disk, and must be kept in sync with the parent. The trade-off is that they isolate the actual filesystem, not just the planning state. Two workspaces can have different files on disk at the same time.

### 9.3 When to use which

Use workstreams for parallel *planning* work: one person researching a security audit while another plans a migration, both against the same live repository. Use workspaces for parallel *execution* work: two phases of different milestones both running executors that need independent source trees. Workstreams share the working copy; workspaces don't.

## 9.4 Execution worktree isolation

A separate feature, `workflow.use_worktrees`, enables automatic worktree creation for *each* executor within a wave. This is a heavier isolation than the workspace model: every parallel executor gets its own git worktree, merges its work back to the parent, and is cleaned up after the wave completes. On projects with aggressive build caches (Rust via Cargo, Scala via sbt) this avoids the lock contention that `--no-verify` commits would otherwise have to work around. The cost is disk space and setup time per wave; the benefit is genuinely independent filesystems for each plan in flight.

# Chapter 10

## Nyquist Validation Layer

### 10.1 The feedback contract

GSD's eighth verification dimension, introduced alongside the Nyquist layer, is a formal feedback contract between requirements and automated tests. For every requirement in `REQUIREMENTS.md`, there must exist at least one automated test that asserts the requirement is met. The planner must declare which tests satisfy which requirements; the verifier checks that the declared tests actually exist and actually pass.

This is the **Nyquist criterion** applied to software verification: if you cannot sample a requirement with a test, you cannot claim to have implemented it. The name is a nod to signal-processing sampling theory — you cannot reconstruct a signal below half the sampling rate — and the philosophical point is the same. Untested claims are unreconstructed.

### 10.2 VALIDATION.md

Each phase directory contains a `XX-VALIDATION.md` file emitted by the `gsd-nyquist-auditor` agent. The file is a matrix: rows are requirements, columns are tests, cells mark which tests cover which requirements and whether they pass. Gaps in the matrix are reported as *validation debt* and surface at the verification gate.

### 10.3 Retroactive validation

The `/gsd-validate-phase` command runs the `nyquist-auditor` against a completed phase, producing a `VALIDATION.md` retroactively. This is the recommended way to add nyquist coverage to projects that adopted GSD mid-stream: run it once per historical phase, inspect the validation debt, and backfill tests until the matrix is full.

## 10.4 Toggle

The feature is controlled by `workflow.nyquist_validation` in `config.json`. Default is `true` (absent = enabled, per GSD's convention). Teams that don't want the overhead can disable it per-project; most teams leave it on because the feedback loop it produces is so directly useful.

# Chapter 11

## Extension Patterns

### 11.1 Agent skill injection

The simplest extension point is the `agent_skills` section of `config.json`. This is a map from agent type (`executor`, `planner`, `researcher`, `verifier`) to a list of prompt blocks that should be injected into that agent's context at spawn time. If your team has a particular testing convention, a specific code style requirement, or a project-wide invariant that every executor must respect, write it as a skill block and register it against `agent_skills.executor`. Every executor spawned thereafter begins with that block in its context.

Skills are ordinary markdown. They can include examples, anti-patterns, or references to other files. The injection is purely additive; GSD does not mutate the agent's base prompt.

### 11.2 Custom hooks

The hook system described in Chapter 1 is extensible. Any script registered in the runtime's `settings.json` under the appropriate event (`PreToolUse`, `PostToolUse`, `statusLine`, `SessionStart`) runs at that event. GSD's own hooks follow a documented contract: read event JSON from stdin, write `hookSpecificOutput` to stdout, exit silently on error, time out stdin reads at three seconds. Custom hooks that follow the same contract will coexist cleanly.

The natural extension points are: additional pre-write guards (e.g., a license-header check), custom statusline formats, and session-start banners that remind the user of project-specific constraints.

### 11.3 Workflow guard as a template

`gsd-workflow-guard.js` is a minimal hook worth reading if you intend to write custom guards. It illustrates the full contract in under 200 lines: stdin parse, event

filter, guard predicate, optional advisory, try/catch wrap. Fork it for your own purposes — e.g., a hook that warns when an edit touches a file listed in a `PROTECTED_FILES` manifest.

## 11.4 Custom GSD commands

Adding a new `/gsd-*` command is as simple as dropping a markdown file into `commands/gsd/` (or `skills/gsd-*/SKILL.md` on Claude Code 2.1.88+) with the appropriate frontmatter. The body of the file is the prompt that the runtime will execute when the user invokes the command. For workflow logic, either embed it directly in the command file or move it to a workflow file under `get-shit-done/workflows/` and reference it from the command with an @-reference.

The installer does not touch user-added files; they survive upgrades. For team distribution, check the added commands into a shared directory and install them alongside GSD via a simple copy step.

# Chapter 12

## Performance Tuning

### 12.1 Model profile optimization

The single biggest lever for performance (and cost) is the model profile. Switching from balanced to budget typically cuts token spend by 60-70% at the cost of some planning quality; switching to max roughly doubles spend for a measurable but small quality gain. Most teams land on balanced and adjust individual roles via `agent_model_overrides`.

A practical rule: use Opus for the planner and verifier; use Sonnet for executors and researchers; use Haiku only for mechanical roles like template filling or statusline generation. The planner's quality disproportionately affects outcomes because a bad plan wastes every executor that runs against it.

### 12.2 Token budget analysis

In a typical medium phase (four plans, two waves, 15-30 files touched), GSD spends tokens roughly as follows: research 15%, planning 25%, plan-checking 10%, execution 40%, verification 10%. The surprising entry is planning at 25%: this reflects the plan-checker loop, which can run up to three revision iterations. If your planning stage is consistently hitting iteration three, the planner is under-powered for your project; upgrade that role to Opus and the iteration count typically drops to one.

### 12.3 Parallelization

Parallelization is free performance when plans are genuinely independent. The wave-analyzer detects independence automatically; your job is to write plans that don't fabricate dependencies. A common anti-pattern is listing every file a plan *might* touch under `<files>`; this over-declares the footprint and causes the wave-analyzer to serialize plans that could have run in parallel. List only files the plan will actually modify.

## 12.4 Worktree isolation

Enabling `workflow.use_worktrees` costs disk space and per-wave setup time but eliminates build-lock contention for projects with aggressive incremental compilers (Cargo, Bazel, sbt, Gradle). On Rust projects with four-plan waves, enabling worktrees has been observed to halve wave wall-clock time because Cargo's target directory is no longer a serialization point.

## 12.5 Context window utilization

The `hooks.context_warnings` section of `config.json` controls the context-monitor thresholds. Defaults are 35% and 25% remaining. Teams that run long interactive sessions (exploratory phases, debug loops) may want to raise these thresholds to trigger earlier; teams running headless CI workloads typically disable them entirely since there's no human to advise.

## Chapter 13

# Team Adoption Guide

### 13.1 Onboarding playbook

A team adopting GSD should expect roughly a two-week ramp. Week one is individual adoption: each engineer installs GSD, runs `/gsd-new-project` against a sandbox, and walks through one discuss-plan-execute-verify loop end-to-end. This produces the muscle memory needed for real use. Week two is team adoption: the team standardizes on a shared `config.json`, agrees on a granularity level, agrees on a branching strategy, and runs its first shared milestone.

The most common failure mode at this stage is *over-granularizing*. New teams split every thought into a separate plan, which multiplies the planning overhead without delivering proportional value. Start coarse and tighten granularity as the team's comfort with plan authoring grows.

### 13.2 Standardizing settings

Commit a `.planning/config.json` to the repository. This file encodes the team's agreed-upon model profile, granularity, branching strategy, nyquist toggle, hook settings, and permission list. Every new clone picks it up automatically; no engineer needs to remember local flags. The only setting that should remain per-user is `user.name` for branch templates.

### 13.3 Granularity tuning for team size

Granularity is the single most important team-scaling dial. At small team sizes (1-3 engineers), coarse granularity is right: fewer, larger plans, less planning overhead, more execution throughput. At medium team sizes (4-10), medium is right: plans are small enough that two engineers can review in parallel. At large team sizes (10+), fine is right: every change is small enough to review, bisect, and revert independently. Tune via `workflow.granularity` in `config.json`.

## 13.4 Code review integration

The `/gsd-code-review` command runs a cross-AI peer review of a phase's commits. Point it at a completed phase, and it spawns a reviewer agent (typically Opus) against the diff. The reviewer looks for common bug patterns, architectural drift from the PLAN's stated intent, and gaps against the phase goals. Output lands in `.planning/phases/XX-phase/XX-REVIEWS.md` and feeds back into the planner if gap-closure is needed.

Teams that practice human peer review should treat `/gsd-code-review` as a pre-review filter: run it before opening the pull request, fix the findings, then ask a human to review what remains. This dramatically reduces the review iteration cycle.

## 13.5 Documentation generation

`/gsd-docs-update` runs the `gsd-doc-writer` and `gsd-doc-verifier` agents in sequence against the repository, producing up-to-date technical documentation from source. The writer produces drafts; the verifier checks them against the actual code, flagging claims that don't match reality. Teams use this to keep README files, API docs, and CONTRIBUTING guides in sync with an evolving codebase without human toil.

## 13.6 `/gsd-milestone-summary` for onboarding

After a milestone completes, `/gsd-milestone-summary` produces a concise narrative of what shipped, why, and what was learned. Feed this to new hires as the canonical project history — it's more honest than a curated CHANGELOG and more readable than a git log.

## 13.7 `/gsd-profile-user` for personalized workflow adaptation

The `/gsd-profile-user` command runs a behavioral profiler against a developer's session history and produces a `USER-PROFILE.md` describing how they work: their response cadence, their preferred granularity, their stop points, the decisions they tend to delegate, and the ones they tend to own. GSD uses this profile to calibrate prompt tone per developer. Teams can opt in team-wide for a consistent experience per engineer, or per-project for a single high-velocity developer. The feature is opt-in and the profile is stored under `.planning/profile/` locally, never transmitted.

# Appendix A

## Security Reference

### A.1 Deny list patterns

The recommended deny list, expanded from Chapter 5 Layer 5, covers the file classes most commonly involved in credential exfiltration incidents:

```
{
  "permissions": {
    "deny": [
      "Read(.env)",
      "Read(.env.*)",
      "Read(**/.env)",
      "Read(**/.env.*)",
      "Read(**/credentials.json)",
      "Read(**/credentials)",
      "Read(**/*.pem)",
      "Read(**/*.key)",
      "Read(**/*.p12)",
      "Read(**/*.pfx)",
      "Read(**/id_rsa)",
      "Read(**/id_ed25519)",
      "Read(**/id_ecdsa)",
      "Read(**/.aws/**)",
      "Read(**/.ssh/**)",
      "Read(**/.gcp/**)",
      "Read(**/.azure/**)",
      "Read(**/secrets/**)",
      "Read(**/private/**)",
      "Read(**/*.kdbx)",
      "Read(**/*.keychain)"
    ]
  }
}
```

This list is not exhaustive. Teams should audit their repositories for project-specific secret locations and extend the list accordingly. A common addition is "Read(\*\*/terraform.tfstate\*)" for Terraform users, and "Read(\*\*/\*.sops.yaml)" for teams using SOPS.

## A.2 Runtime hook configuration

To enable the workflow guard (advisory, opt-in), set in `.planning/config.json`:

```
{
  "hooks": {
    "workflow_guard": true,
    "context_warnings": {
      "warning_threshold": 0.35,
      "critical_threshold": 0.25,
      "debounce_tools": 5
    }
  }
}
```

The prompt guard runs by default and cannot be disabled from configuration; uninstall or replace the hook file if you need to disable it for development purposes.

## A.3 CI scanner integration

The CI injection scanner at `tests/prompt-injection-scan.test.cjs` runs as part of the standard test suite. For teams that want to run it standalone in a pre-merge check, invoke the shell wrapper:

```
./scripts/prompt-injection-scan.sh
```

Exit code zero indicates no high-severity findings. Non-zero indicates at least one finding; output enumerates file, pattern, and line. Wire this into your CI system as a required status check on pull requests that modify `.planning/` or any markdown under `docs/`.

## Appendix B

# Configuration Reference

### B.1 Top-level structure

`.planning/config.json` is a JSON object with several top-level sections. The most important keys, with their types, defaults, and effects:

#### Model profile and overrides

Key	Type	Default	Effect
<code>model_profile</code>	string	balanced	Named profile controlling per-role model selection. Valid: max, balanced, budget, inherit.
<code>agent_model_overrides</code>	object	{}	Per-agent-type overrides that win over the profile. Keys are agent type names, values are model identifiers.
<code>context_window</code>	integer	200000	Declared context window size. When $\geq 500,000$ , enables adaptive context enrichment for 1M-class models.

#### Workflow toggles

Key	Type	Default	Effect
<code>workflow.granularity</code>	string	medium	Plan splitting granularity. Valid: coarse, medium, fine. Affects planning task-splitting heuristics and summary template selection.
<code>workflow.mode</code>	string	standard	Overall workflow mode. Valid: standard, fast, autonomous. Fast and autonomous reduce gates.
<code>workflow.branching</code>	string	none	Git branching strategy. Valid: none, phase, milestone.
<code>workflow.branch_template</code>	string	<code>gsd/phase-{number}-{slug}</code>	Template for branch names. Variables: {number}, {slug}, {date}, {user}.

Key	Type	Default	Effect
<code>workflow.merge_strategy</code>	string	merge	How to merge on completion. Valid values are merge, squash, rebase.
<code>workflow.use_worktrees</code>	boolean	false	Enable per-executor worktree isolation within waves.
<code>workflow.nyquist_validation</code>	boolean	true	Enable the 8th verification dimension (test-to-requirement mapping). Absent = enabled.
<code>workflow.parallel_execution</code>	boolean	true	Enable wave-parallel executor spawning. Absent = enabled.
<code>workflow.plan_checker_max_iterations</code>	integer	3	Maximum plan revision iterations before halting for human review.

## Hook configuration

Key	Type	Default	Effect
<code>hooks.workflow_guard</code>	boolean	false	Enable the file-editor Opt-in. Produces no warnings.
<code>hooks.context_warnings.warning_threshold</code>	number	0.35	Remaining-context threshold to inject the WARNING hook.
<code>hooks.context_warnings.critical_threshold</code>	number	0.25	Remaining-context threshold to inject the CRITICAL hook.
<code>hooks.context_warnings.debounce_tools</code>	integer	5	Tool-use count before showing warnings at the same time.
<code>hooks.prompt_guard_log</code>	string	<code>.planning/security/prompt-guard.log</code>	Path for prompt-guard logs.

## Agent skill injection

Key	Type	Default	Effect
<code>agent_skills.executor</code>	array	[]	Prompt blocks injected into every executor agent at spawn. Each entry is a path to a markdown file.
<code>agent_skills.planner</code>	array	[]	Same, for planner agents.
<code>agent_skills.researcher</code>	array	[]	Same, for researcher agents.
<code>agent_skills.verifier</code>	array	[]	Same, for verifier agents.

## Permissions (delegated to runtime)

Permission configuration lives in the runtime's own `settings.json` rather than GSD's `config.json`. The keys that matter for GSD users are `permissions.allow` (an allow-list of tool patterns approved without prompting) and `permissions.deny` (the hard block list, including the secret deny list from Appendix A). Claude Code, OpenCode, and Cursor all honor this format.

## B.2 Convention: absent = enabled

GSD follows an **absent = enabled** convention for workflow feature flags. If a boolean key is missing from *config.json*, it defaults to true. Users explicitly disable features they don't want; they do not need to enable defaults. This applies to `workflow.nyquist_validation`, `workflow.parallel_execution`, and other boolean toggles. The convention minimizes configuration churn for new projects, which get correct defaults automatically, at the cost of a slight learning curve for engineers expecting explicit opt-in everywhere.