

GSD

Going Further

Skills, MCP, and Multi-Repo Workflows

What This Guide Covers

Three advanced GSD topics, each delivered through a complete walkthrough you can follow at the keyboard:

- **Skills & Slash Commands** — the SKILL.md format, the 12-runtime install matrix, and how to inject custom domain knowledge into GSD's subagents.
- **MCP Integration** — connecting GSD to external tools via the Model Context Protocol, with full walkthroughs for Google Stitch UI generation and schema-aware planning via a Postgres MCP.
- **Multi-Repo Workflows** — workstreams, workspaces, the STATE.md ownership rule, and patterns for teams running GSD across microservice fleets.

Tibsfox

For GSD v1.35.0

Audience: GSD users ready to customize, integrate, and scale

Released under CC BY-SA 4.0.

Authored by Tibsfox. GSD itself is MIT-licensed — see gsd-build/get-shit-done.

Contents

I Skills & Slash Commands	5
1 How GSD Actually Works — Skills Anatomy	6
1.1 Finding the Skills on Disk	6
1.2 The SKILL.md Format	7
1.3 How Trigger Descriptions Work	8
1.4 The Installer’s Role	8
1.5 Skills versus Legacy Slash Commands	9
1.6 What You Can Do With This Knowledge	9
2 Skills Across Runtimes	10
2.1 The Compatibility Matrix	10
2.2 How the Installer Transforms	11
2.2.1 Skills into Prompts (Copilot)	11
2.2.2 Skills into Rules (Cline)	11
2.2.3 Skills into TOML + Markdown (Codex)	11
2.3 Runtime-Specific Gotchas	11
2.3.1 Cline Has No Slash Commands	11
2.3.2 Copilot Needs Both Halves	12
2.3.3 Codex Uses TOML Pairs	12
2.3.4 Antigravity Has Two Install Dirs	12
2.4 Non-Claude Model Profiles	12
2.5 Verifying an Install	13
3 Agent Skills Injection	14
3.1 The Concept	14
3.2 Configuration Syntax	15
3.3 Which Agent Types Accept Skills	15
3.4 Writing a Skill Directory	15
3.5 How Injection Affects Behavior	16
3.6 A Minimal Example	16
3.7 When to Reach for This Tool	17
4 Scenario — Custom Research Skill	18

II MCP Integration	22
5 MCP Fundamentals for GSD Users	23
5.1 What MCP Actually Is	23
5.2 Why MCP Matters for GSD Specifically	24
5.3 Transport Types: stdio and HTTP	24
5.3.1 stdio Transport	24
5.3.2 HTTP Transport (Streamable HTTP)	24
5.4 How Discovery Works	25
5.5 A Short History and Why It Matters	25
5.6 What MCP Is Not	26
6 Configuring MCP Servers	27
6.1 Adding Servers via the CLI	27
6.2 Project-Scoped Configuration: .mcp.json	27
6.3 User-Scoped Configuration: ~/.claude.json	28
6.4 Project vs User Scope: How to Decide	28
6.5 Environment Variables in Config	29
7 Scenario — Google Stitch + GSD	31
8 Scenario — Database MCP + GSD Planning	37
9 Building MCP-Aware GSD Workflows	40
9.1 Combining Agent Skills with MCP	40
9.2 The GSD Plugin Format (Issue #1883)	41
9.3 Failure Modes: When an MCP Server is Unavailable	42
9.4 Observability: Knowing What Your MCP Servers Did	42
9.5 Version Drift: What Happens When a Server Changes	43
III Multi-Repo Workflows	45
10 Workstreams vs. Workspaces — When to Use Which	46
10.1 The Problem: Shared Planning State	46
10.2 Workstreams: One Repo, Many Timelines	47
10.3 Workspaces: Many Repos, One Plan	47
10.4 Decision Table	48
10.5 One More Consideration: How Long Does the Work Live?	49
11 Scenario — Monorepo with Independent Apps	50
12 Scenario — Microservice Fleet	53
13 Worktree Isolation Deep Dive	56
13.1 What workflow.use_worktrees: true Does	56
13.2 The STATE.md Ownership Rule	57
13.3 What Agents Actually Write: SUMMARY.md	57
13.4 Post-Wave Aggregation	57
13.5 The git clean Prohibition	58

13.6 Toggling Worktrees Off	58
14 Team Patterns for Multi-Repo GSD	60
14.1 Standardizing Config Across Repos	60
14.2 MCP Server Scoping, Restated	61
14.3 Workstream Naming Conventions	61
14.4 The project_code Field	62
14.5 Ticket-Based Phase Identifiers	62
A Runtime Compatibility Matrix	64
B MCP Server Quick-Reference	66
C Workspace Command Cheat Sheet	67

Part I

Skills & Slash Commands

Chapter 1

How GSD Actually Works --- Skills Anatomy

Most GSD users never look under the hood. They install the tool, type `/gsd:new-project`, and watch Claude Code do the rest. That is exactly how it is supposed to feel. But the moment you want to bend GSD to fit your team, your domain, or your personal taste, the black box stops being a feature and starts being an obstacle.

This chapter opens the box. By the end of it you will know where GSD stores its skills on disk, what a single skill file looks like, how Claude Code decides which skills to activate, and why the installer ships two different layouts depending on which version of Claude Code you have. None of this requires source code spelunking — everything GSD installs is plain markdown, which means you can read it, diff it, copy it, and edit it with the same tools you already use.

1.1 Finding the Skills on Disk

On a Claude Code 2.1.88 or newer install, GSD writes its skills into your per-user Claude configuration directory. On Linux and macOS that lives at `~/.claude/skills/`; on Windows it is `%USERPROFILE%\claude\skills\`. The GSD installer creates one sub-directory per command, each prefixed with `gsd-`, and each containing a single `SKILL.md` file:

```
~/.claude/skills/|—
gsd-new-project/|
  └─ SKILL.md |—
gsd-plan-phase/|
  └─ SKILL.md |—
gsd-execute-phase/|
  └─ SKILL.md |—
gsd-verify-work/|
  └─ SKILL.md |—
gsd-progress/|
  └─ SKILL.md |—
gsd-help/|
```

```
└─ SKILL.md ─┘  
... (30+ more)
```

Go ahead and navigate there now. Run `ls ~/.claude/skills/` on your own machine. You should see a long list of `gsd-*` directories plus any other skills Claude Code has picked up from other tools or from your own hand-authored experiments. This is not a sandbox — it is your real home directory, and those markdown files are the exact instructions Claude reads when you invoke a GSD command.

You can read any GSD skill file. They are plain markdown — no compilation, no opaque binaries, no generated code. If you are ever unsure what a `/gsd:*` command is *actually* going to do, open its `SKILL.md` in your editor and read it. That is the entire contract.

1.2 The SKILL.md Format

Every skill file Claude Code 2.1.88 and later consumes follows the same shape: a YAML frontmatter block, a heading, and an instruction body. The frontmatter carries two required keys — `name` and `description` — and the body is markdown prose that becomes the instruction set the model sees when the skill activates.

Here is the minimum viable skill file:

```
---  
name: gsd-progress  
description: "Show GSD project progress and current phase status"  
---  
# GSD Progress Report  
  
When the user asks about the current state of the project, read  
`.planning/STATE.md` and `.planning/ROADMAP.md`. Produce a compact  
summary containing:  
  
1. The current milestone (from ROADMAP.md).  
2. The active phase (from STATE.md).  
3. The most recent three completed work items.  
4. Any open blockers recorded under the "Blockers" heading.  
  
Render the output as a markdown table with two columns: "Item" and  
"Status". Do not fabricate phase data --- if STATE.md is missing, say  
so and suggest running `/gsd:new-project` to initialize it.
```

That is a complete, working skill. Drop that file into `~/.claude/skills/gsd-progress/SKILL.md`, reopen Claude Code, and typing *“show me the GSD progress”* will surface it as a candidate skill. The `name` field is the stable identifier — it also determines the slash-command alias (`/gsd:progress` maps to `gsd-progress`) on runtimes that expose slash commands. The `description` field is where the real magic lives, and it deserves its own section.

1.3 How Trigger Descriptions Work

Claude Code does not load every skill on disk into every request. That would blow the context window inside of a dozen commands. Instead the runtime keeps a lightweight manifest of all installed skills — just their names and descriptions — and when a new user message arrives, it pattern matches the message against those descriptions to decide which skills to promote into the full context. Only promoted skills have their instruction body loaded.

The description is therefore the *activation contract*. A vague description like "Help with GSD" will match almost everything and almost nothing usefully. A concrete one like "Show GSD project progress and current phase status" will match phrases like "where are we on the project", "what phase are we in", or "give me a status update" without swamping unrelated requests.

When you author your own GSD-adjacent skills, invest time in the description line. Use action verbs, name the artifact you operate on, and include a few of the phrasings users actually say out loud. A good description is not a summary of what the skill does — it is a list of *triggers* that should fire it.

The frontmatter is strict YAML. Unquoted colons in the description will break the parser and silently disable the skill. When in doubt, wrap the description in double quotes. This is the single most common reason a hand-authored skill "doesn't work".

1.4 The Installer's Role

GSD ships as an npm package. When you run `npx gsd-build install`, the installer does three things in sequence:

1. **Detect the runtime.** It looks at which agent CLIs are installed on the current machine — Claude Code, Codex, Cursor, Windsurf, and so on — and also at the *version* of Claude Code it finds. The 2.1.88 cutoff matters: earlier versions do not understand the skills directory layout at all.
2. **Read its source skill definitions.** GSD carries its command definitions in a single canonical form inside the npm package. These are the master copies.
3. **Transform and write.** For each detected runtime, the installer rewrites the master skill into whatever format that runtime actually consumes, then drops the result into the right directory. One master, many derivatives.

This architecture is why the `same /gsd:plan-phase` command behaves identically whether you are driving Claude Code, Codex, or Cursor — the installer is doing the translation work up front so the user-facing surface is uniform.

1.5 Skills versus Legacy Slash Commands

Before Claude Code 2.1.88, Claude Code had no notion of skills. The only way to ship reusable instructions was as a *slash command*: a markdown file living at `~/.claude/commands/gsd/[name].md` that the user invoked by literally typing its name. There was no description field, no trigger matching, no auto-activation — the user had to know the command existed and type it explicitly.

GSD still supports this old world. If the installer detects a pre-2.1.88 Claude Code install, it writes legacy slash commands instead of skills:

```
~/.claude/commands/gsd/|—  
new-project.md|—  
plan-phase.md|—  
execute-phase.md|—  
verify-work.md|—  
...  
|
```

You interact with both layouts the same way from the user side — `/gsd:plan-phase` works whether it resolves to a skill or a legacy command — but under the hood the runtime treats them differently. Skills are auto-activating and context-aware; legacy commands only fire on an explicit invocation. If you upgrade Claude Code from an older version, run `npx gsd-build install` again and the installer will migrate you forward automatically, leaving the legacy directory in place so nothing breaks mid-session.

1.6 What You Can Do With This Knowledge

Once you internalize that GSD skills are just markdown on disk, a long list of previously scary tasks becomes trivial. You can diff two installs to see what changed between GSD releases. You can keep a local patch directory and re-apply your own edits after an upgrade. You can fork a skill, rename it to `gsd-plan-phase-health`, tailor it to your domain, and have both the generic and the specialized version coexist in the same directory. You can hand-write skills that call GSD primitives without waiting for upstream to ship them. Everything is text, everything is readable, and nothing is hidden.

The rest of this part walks through each of those capabilities in detail, starting with the compatibility matrix that determines where your skills actually land.

Chapter 2

Skills Across Runtimes

GSD is not a Claude Code exclusive. The installer knows about a dozen different agent runtimes, each with its own directory layout, its own file format, and its own opinions about how instructions should be delivered to the model. This chapter is the reference you will reach for when a GSD command that works on one machine appears to be missing on another. The short answer is almost always *“it installed fine, but it lives somewhere else on this runtime.”*

2.1 The Compatibility Matrix

The table below lists every runtime the GSD installer currently supports, the format it emits for that runtime, the directory where the output lands, and the command you can run to confirm the install succeeded.

Runtime	Install Format	Directory (Global)	Verify Command
Claude 2.1.88+	Code	skills/gsd-*/SKILL.md ~/.claude/skills/	/gsd:help
Claude (legacy)	Code	commands/gsd-*.md ~/.claude/commands/	/gsd:help
Codex		skills/gsd-*/SKILL.md ~/.codex/skills/	\$gsd-help
Copilot		prompts + agents ~/.github/	/gsd:help
Cursor		skills/gsd-*/SKILL.md ~/.cursor/	/gsd:help
Windsurf		markdown transform ~/.windsurf/	/gsd:help
OpenCode		config file ~/.config/opencode/	/gsd-help
Gemini CLI		skills ~/.gemini/	/gsd:help
Antigravity		skills ~/.gemini/antigravity/ or ~/.agent/	/gsd:help
Cline		.clinerules ~/.cline/	rules auto-loaded
Augment		skills transform ~/.augment/	/gsd:help
Qwen Code		skills (open standard) ~/.qwen/skills/	/gsd:help

Twelve runtimes, twelve directory layouts. Four of them (Claude Code 2.1.88 and later, Codex, Cursor, and Qwen Code) converge on the same skills-with- SKILL.md shape, which is the format this book treats as canonical. The other eight each

require the installer to do some amount of translation work, and understanding those translations is the key to debugging any cross-runtime weirdness you run into.

2.2 How the Installer Transforms

The installer maintains a single internal representation for each GSD command — call it the “source form.” When a runtime requires a different format, the installer applies a translation pass. Here are the three most important translations.

2.2.1 Skills into Prompts (Copilot)

GitHub Copilot does not have a skills directory. It has a `prompts/` directory for standalone prompt files and an `agents/` directory for role definitions, and the two work together. The installer splits each GSD skill in half: the instruction body becomes a prompt file under `~/.github/prompts/gsd/`, and the trigger metadata plus any agent role information becomes an agent definition under `~/.github/agents/`. Invoking `/gsd:plan-phase` in Copilot fires both sides simultaneously.

2.2.2 Skills into Rules (Cline)

Cline takes the opposite approach. It has no concept of slash commands at all — instead, it reads a `.clinerules` file and treats the entire contents as a persistent instruction prefix injected into every request. The installer concatenates every GSD skill body into a single `.clinerules` file and drops it at `~/.cline/.clinerules`. You will not find a `/gsd:plan-phase` slash command in Cline because there are no slash commands to find; the rules simply load automatically and Cline recognizes the GSD phrasings when you use them conversationally.

2.2.3 Skills into TOML + Markdown (Codex)

Codex uses a paired-file convention: a `.toml` file declares the skill metadata, and a matching `.md` file alongside it carries the instruction body. The installer emits both halves. If you open `~/.codex/skills/gsd-plan-phase/` you will see a `skill.toml` with the name and description fields, plus a `SKILL.md` with the prose. Codex also uses a different invocation prefix: `$gsd-help` instead of `/gsd:help`, which is worth memorizing if you move between machines.

2.3 Runtime-Specific Gotchas

Even when you know the translation rules, a few runtimes have quirks worth calling out explicitly.

2.3.1 Cline Has No Slash Commands

If a new Cline user complains that `/gsd:plan-phase` “isn’t working,” they have usually not done anything wrong — Cline literally does not support slash commands.

The fix is not to reinstall. The fix is to describe what you want in natural language (“*plan the next phase*”) and let the injected `.clinerules` do its job.

2.3.2 Copilot Needs Both Halves

The prompt and agent files Copilot writes work together, and if only one half lands (for example, because the installer was interrupted or because `~/github/agents/` already contained a conflicting file), the GSD command will misbehave in subtle ways. Re-running `npm run gsd-build install --force` will overwrite any half-written state and is the standard fix.

2.3.3 Codex Uses TOML Pairs

If you hand-edit a Codex skill, remember that the metadata lives in the `.toml` file, not in frontmatter. Editing the `SKILL.md` body to change the description field will have no effect — Codex never reads frontmatter. Edit the `.toml`.

2.3.4 Antigravity Has Two Install Dirs

Antigravity supports both a global install at `~/gemini/antigravity/` and a project-local install at `./.agent/`. The installer picks global by default but will honor the `--local` flag to install into the current project. This is the only runtime where GSD supports per-project scoped command installs at the runtime layer; for every other runtime, per-project customization happens through the `agent_skills` mechanism described in the next chapter.

2.4 Non-Claude Model Profiles

Most GSD subagents (the executor, planner, researcher, and verifier roles) are built on Claude Code’s subagent spawn primitive, which defaults to whichever Anthropic model the parent session is running. That is usually what you want — consistency across the session — but it is a footgun the first time you run GSD inside an OpenRouter-fronted Claude Code or on a locally hosted model.

The problem: even when the parent session is routed through OpenRouter, a spawned subagent may try to call Anthropic directly unless you explicitly tell it not to. You end up paying OpenRouter for the parent and Anthropic for every subagent spawn simultaneously.

GSD ships an `inherit` profile for exactly this case. When you select the `inherit` profile (via `/gsd:set-profile inherit` or by editing `.planning/config.json`), every spawned subagent will reuse the parent session’s model configuration instead of defaulting to Anthropic. If the parent is on OpenRouter, the subagents are on OpenRouter. If the parent is on a local Ollama model, the subagents are too.

If GSD subagents call Anthropic models and you are paying through OpenRouter, switch to the `inherit` profile or you will see double-billing. Run `/gsd:set-profile inherit` once per project.

2.5 Verifying an Install

Regardless of runtime, the fastest way to confirm GSD is actually installed is to run the `/gsd:help` command (or its runtime-specific equivalent from the matrix). It is a skill like any other, and if it responds with the GSD command listing, everything else is wired up correctly. If it does not respond, the first diagnostic step is to `ls` the install directory for your runtime and confirm the `gsd-*` entries exist. Ninety percent of install problems are really “I am on a runtime I did not expect to be on” problems, and the matrix is there to cut that debugging time down to seconds.

Chapter 3

Agent Skills Injection

So far everything in this Part has been about *global* skills — the files GSD installs into your per-user directory that apply to every project you work on. That is the right default for the command layer. But GSD also has a sharper tool for per-project customization, one that does not require touching the global directory at all: `agent_skills`.

`agent_skills` is the mechanism by which a single project teaches its GSD subagents things about its own domain. Those teachings live inside the project, travel with it through version control, and apply only while you are working in that project. No other project sees them. No global state gets polluted.

3.1 The Concept

When GSD spawns a subagent — an executor running a phase, a planner drafting the next set of work items, or a researcher collecting domain information — it builds the subagent's system prompt from a stack of pieces: the subagent's role definition, the current project state, the task at hand, and optionally, one or more *injected skill blocks* drawn from local project directories.

Those injected blocks arrive in the subagent's prompt as an `<agent_skills>` section. From the subagent's point of view, they are just more instructions — indistinguishable from the built-in role definition except that they came from a project-local directory rather than from the GSD source tree. This is pure prompt augmentation. There is no plugin system, no sandbox, no new execution path. The injected content simply becomes part of what the subagent sees.

That simplicity is the feature. You can teach a GSD researcher about FDA endpoints the same way you would teach a human researcher about them: by handing over a short written brief. The researcher reads the brief, integrates it into its working knowledge, and uses it for the duration of the task.

3.2 Configuration Syntax

`agent_skills` is configured in the project's `.planning/config.json` file. The key is `agent_skills`, and its value is an object mapping subagent type names to arrays of directory paths. Each path is resolved relative to the project root.

```
{
  "agent_skills": {
    "executor": ["/skills/domain-knowledge/"],
    "planner": ["/skills/planning-rules/"],
    "researcher": ["/skills/research-sources/"]
  }
}
```

When a subagent of a given type spawns, the installer walks the directories listed under that type, reads every `.md` file it finds, concatenates the contents, and wraps the result in an `<agent_skills>` block inside the subagent's prompt. Files are read in lexicographic order, so prefixing names with `01-`, `02-`, and so on gives you explicit control over the final ordering when order matters.

3.3 Which Agent Types Accept Skills

Not every subagent type honors `agent_skills`. As of the current release the supported types are:

- **executor** — the phase-running worker. Injected skills here shape how code gets written, which conventions get followed, and which libraries get preferred. Good for “our codebase uses X pattern” style rules.
- **planner** — the task decomposition agent. Injected skills here shape how phases get broken into work items and what kinds of safety rails the planner enforces. Good for “never do X without Y” style rules.
- **researcher** — the information-gathering agent. Injected skills here shape which sources get consulted and what domain vocabulary gets used in the resulting `RESEARCH.md`. Good for teaching the researcher about APIs, standards, databases, and terminology it would not otherwise know.

The verifier, documenter, and other specialized subagent types currently use a fixed instruction set and do not read `agent_skills`. That may change in a future release; for now, if you need to customize verification behavior, you do it by editing the global skill file directly.

3.4 Writing a Skill Directory

A skill directory is just a directory. Create it anywhere under your project root (convention is `./skills/[name]/`), drop one or more `.md` files in it, and list the directory in `agent_skills`. That is the whole contract.

The markdown files inside do not need frontmatter. They do not need a name or description field. They just need to contain the instructions you want the subagent to read. You can write them in whatever style feels natural — prose, bullet lists, labeled sections, annotated code examples — and the subagent will parse them the same way it parses any other instruction block.

The reason naming is descriptive (rather than formal) is that every `.md` file's content gets concatenated together at spawn time. The subagent does not see file boundaries; it sees one long block of text. File names serve only two purposes: ordering (lexicographic) and readability for you when you come back to the directory in six months.

3.5 How Injection Affects Behavior

The injected `<agent_skills>` block lives inside the subagent's prompt for the entire duration of its task. That means:

- The instructions persist across every tool call the subagent makes. A researcher that spawns a web search, then a file read, then another web search still has the FDA brief visible through all three calls.
- The instructions are visible to the model at every decision point. When the model is picking which URL to fetch or which section to write first, it is weighing those choices against the injected skill content.
- The instructions do not persist beyond the task. Once the subagent exits, the injected block goes with it. The next subagent spawn rebuilds the prompt from scratch, picking up any edits you made to the skill files in between.

That last point is important: you can edit a skill file mid-session and the next subagent spawn picks up the change immediately. There is no rebuild step, no cache to invalidate, no installer to re-run.

3.6 A Minimal Example

Suppose you are building a product that must never merge to main without first running an integration test. You want the GSD planner to refuse to generate a plan that skips this step. Here is the full contents of `./skills/planning-rules/no-untested-merges.md`:

```
# Merge Safety Rule

Never produce a plan whose final step is a merge to `main`, `master`,
or any protected branch unless the plan also contains an explicit
integration test step that runs BEFORE the merge and must pass.

If a user asks for a plan that would merge without testing, respond
with a plan that adds the missing test step and a one-line note
explaining why.

Applies to: all merge-containing phases, all branches matching
`main`, `master`, `release/*`, or `prod`.
```

Six lines of instruction. That is enough. With this file in place and planner pointing at `./skills/planning-rules/` in `agent_skills`, every subsequent `/gsd:plan-phase` invocation in this project will generate plans that include a test step before any merge, and will add a short justification when the user's request would have skipped one.

The planner did not need to be recompiled. The GSD installer did not need to be rerun. You wrote six lines of markdown, saved the file, and the next planner spawn behaved differently. That is the entire feature.

`agent_skills` are project-local. They are defined in `.planning/config.json`, they reference paths under the project root, and they do not affect any other project on the same machine. Commit the skill directories along with your code so the next developer who clones the repo gets the same behavior automatically.

3.7 When to Reach for This Tool

`agent_skills` is the right tool whenever your customization need has three properties: it is specific to one project, it is worth writing down, and it would benefit from being seen by a particular subagent role on every spawn. Domain vocabulary, compliance rules, in-house API conventions, preferred libraries, house coding standards, data-handling constraints — all of these fit naturally.

It is the wrong tool for global rules (those belong in a hand-authored global skill), for one-off asks (those belong in the prompt itself), and for rules the whole team has not yet agreed on (those belong in a discussion, not a config file). Within its sweet spot, though, it is the single most powerful lever GSD gives you for shaping subagent behavior — and the next chapter walks through a complete, end-to-end use of it.

Chapter 4

Scenario --- Custom Research Skill

You are three weeks into building a small health-tech application — a patient intake form that needs to map its fields cleanly to the FHIR Patient resource, check drug interactions against an authoritative source, and keep one eye on the 510(k) clearance pathway in case the product ever grows into a regulated medical device. You are using GSD to manage the work. Planning, execution, and verification all run smoothly. But the research phase keeps coming up short.

Here is what you want to fix, and how `agent_skills` fixes it.

The problem.

You ran `/gsd:research-phase` yesterday on the topic *“data sources for drug interaction checking.”* The researcher subagent did its job: it spawned, ran a batch of web searches, read a handful of Wikipedia pages and generic developer blogs, and wrote a tidy `RESEARCH.md` file into `.planning/research/`. Tidy, but wrong for your use case. The output was full of generic phrases like *“drug interaction APIs such as those offered by various healthcare data vendors,”* hand-waving references to *“the standard medical data format,”* and not a single mention of the FDA’s openFDA Drug Event API, the RxNorm concept hierarchy, or the DrugBank identifier system. The section on data formats talked about JSON generically but never named FHIR, never mentioned the MedicationStatement resource, and never hinted that a serious health-tech product would need to handle HL7 interop.

This is the researcher working exactly as designed. The problem is not that it is bad at research; the problem is that it has no reason to prefer FDA over DrugBank over a random consumer health blog, because nobody told it which sources are authoritative in your domain. You are going to tell it now.

Step 1: Create the skill directory.

Start by carving out a home for the domain brief. GSD convention is to put project-local skill directories under `./skills/`, so make one there with a descriptive name:

```
mkdir -p ./skills/health-research/  
touch ./skills/health-research/SKILL.md
```

Open that new `SKILL.md` in your editor and write a real brief. This is the file you will hand to every future researcher spawn, so make it concrete. Name the endpoints, name the resources, name the regulations. Here is the full contents of the file we used for this project:

```
# Health-Tech Research Brief

When researching any topic related to drug data, patient records,
clinical workflows, or medical devices, prefer the following
authoritative sources over generic web results. Always cite them
by name in the final RESEARCH.md.

## Drug data
- FDA openFDA Drug Event API:
  https://api.fda.gov/drug/event.json
  Use for adverse event lookups and label data.
- RxNorm (NLM):
  https://rxnav.nlm.nih.gov/REST/
  Use for normalized drug name resolution.
- DrugBank (academic tier):
  https://go.drugbank.com/
  Use for interaction data; cite the DrugBank ID.

## Clinical trials and evidence
- ClinicalTrials.gov search API:
  https://clinicaltrials.gov/api/
  Use for active and completed trials, inclusion criteria,
  and results summaries.

## Data model
- FHIR R4 resource definitions:
  https://hl7.org/fhir/R4/resourcelist.html
  When modeling patient data, reference the exact FHIR resource
  and field names (e.g., Patient.identifier, Observation.code,
  MedicationStatement.medicationCodeableConcept).

## Regulatory
- HIPAA Privacy Rule: treat all PHI fields as requiring encryption
  at rest and in transit. Never store unmasked PHI in logs or
  telemetry.
- 510(k) Premarket Notification: if a feature could classify as
  a medical device, flag it and link to
  https://www.fda.gov/medical-devices/premarket-submissions/
  premarket-notification-510k
```

Notice what this file is *not*. It is not a summary of everything there is to know about health tech. It is not a tutorial. It is not a policy document. It is a short, dense list of *“here are the sources to use and the vocabulary to use when writing up your findings.”* Twenty-some lines is plenty. The researcher is already capable of doing research; you are just handing it a better rolodex.

Step 2: Configure agent_skills.

Open `.planning/config.json` and add a researcher entry to the `agent_skills` object. If the object does not yet exist, create it. Here is the relevant snippet with the new entry in place:

```
{
  "project_name": "health-intake",
  "profile": "inherit",
```

```
"agent_skills": {  
  "researcher": ["/skills/health-research/"]  
}
```

Save the file. That is the entire configuration step. No installer rerun, no restart, no validation pass. The GSD runtime reads `.planning/config.json` on every subagent spawn, so the next time a researcher starts, it will pick up the new skill directory.

Step 3: Run `/gsd:research-phase`.

Re-run the same research phase you ran yesterday. This time the planner subagent reads your request, kicks off a researcher spawn, and the GSD runtime assembles the researcher's prompt: role definition, task description, project state, and now — because you wired it up — an `<agent_skills>` block containing your health-tech brief.

```
/gsd:research-phase "data sources for drug interaction checking"
```

What happens inside the spawn: the researcher reads the brief as part of its initial context, sees the named endpoints (openFDA, RxNorm, DrugBank), and queues those searches first. When it hits web results for generic health blogs, it now has a yardstick to measure them against, and it deprioritizes them. When it comes time to write up findings in `RESEARCH.md`, the vocabulary from the brief — MedicationStatement, Observation.code, HIPAA, 510(k) — shows up in the prose because the researcher saw those terms as part of its working instructions.

Step 4: Verify `RESEARCH.md`.

Open `.planning/research/RESEARCH.md` and confirm the domain knowledge landed. Here is an excerpt from the file our example produced, covering the “Drug data sources” and “Data format” sections:

```
## Drug data sources  
  
Primary: FDA openFDA Drug Event API  
(https://api.fda.gov/drug/event.json). Use `search=` query  
parameter for adverse event lookups; rate limit 240 req/min  
unauthenticated.  
  
Normalization: RxNorm via the NLM RxNav REST endpoint  
(https://rxnav.nlm.nih.gov/REST/). Use `rxcai` lookup to  
collapse brand names to concept IDs before interaction checks.  
  
Interaction data: DrugBank academic tier. Cite DrugBank ID  
(format DBnnnnn) in all interaction records.  
  
## FHIR Patient resource shape  
  
- Patient.identifier (array of Identifier)  
- Patient.name (array of HumanName)  
- Patient.telecom (array of ContactPoint, PHI)  
- Patient.birthDate (date)  
- Patient.address (array of Address, PHI)  
  
PHI fields MUST be encrypted at rest per HIPAA Privacy Rule.
```

Twelve lines, and every one of them is grounded in something the brief told the researcher to care about. Endpoints are cited with real URLs. Resources are named by

their exact FHIR field paths. The HIPAA note is present because the brief flagged it. Compare this to yesterday's output and the difference is not subtle — this reads like something a health-tech engineer would actually hand to a coworker, not a generic tour of the internet.

Step 5: Iterate.

You will not get the brief perfect on the first try. Maybe the researcher missed the SNOMED CT code system, or over-cited DrugBank when RxNorm would have been more appropriate, or under-cited the 510(k) clearance pathway because your wording was too hedged. The fix is straightforward: open `./skills/health-research/SKILL.md` again, sharpen the instructions, save the file, and re-run `/gsd:research-phase`. The next researcher spawn picks up the edits without any rebuild step, because `agent_skills` reads live from disk on every spawn.

If you find that a single monolithic file is getting unwieldy, split it into multiple markdown files inside the same directory. Name them `01-sources.md`, `02-fhir.md`, `03-regulatory.md`, and GSD will concatenate them in lexicographic order. The subagent still sees one continuous block, but you get to reason about the pieces separately.

When you are happy with the brief, commit it. The `./skills/health-research/` directory and the updated `.planning/config.json` both belong in version control, and every developer who clones the repo will get the same researcher behavior automatically — no setup steps, no onboarding friction, no tribal knowledge about “the way we do research on this project.” The knowledge is written down, and it travels with the code.

Skill files are live. Edit and re-run — there is no rebuild step. If a GSD subagent is missing context, the fastest path to a fix is almost always “write it into the skill file and spawn again.”

You have now seen the full shape of GSD's skills system: global skills that live in your per-user directory and apply everywhere, legacy slash commands for older Claude Code versions, twelve runtime targets that all converge on the same user experience, and project-local `agent_skills` that let you teach GSD subagents anything your domain needs them to know. Part II picks up from here and looks at agents and teams — the mechanism by which GSD composes subagents into larger, coordinated work units.

Part II

MCP Integration

Chapter 5

MCP Fundamentals for GSD Users

If Part I was about teaching GSD to speak your project’s dialect, Part II is about teaching GSD to reach outside of itself. Most of the real work developers do involves tools: design tools, databases, ticket systems, cloud consoles, monitoring dashboards, that one custom script your team lead wrote three years ago that nobody fully understands. Claude Code can talk to all of them — but only if a bridge exists. That bridge is MCP.

5.1 What MCP Actually Is

MCP — the Model Context Protocol — is an open standard, published by Anthropic, for connecting AI assistants to external tools and data sources. At its heart it is a small, stable JSON-RPC interface. A program that implements MCP is called an *MCP server*. A program that connects to one is called an *MCP client*. Claude Code is a client. Almost everything else in this chapter is a server.

Each MCP server exposes a handful of *tools*. A tool has a name, a JSON schema describing its arguments, and a function that runs when Claude Code invokes it. During a conversation, Claude Code sends a list-tools request to every connected server at startup, merges the results into its own tool surface, and from that point on, calling an MCP tool looks identical to calling a built-in one. The assistant does not know or care whether `read_file` is implemented in Claude Code itself or served by a filesystem MCP running on your laptop.

This is a much bigger deal than it sounds. Before MCP, every integration between an assistant and an outside service had to be hand-coded, vendor by vendor, with a bespoke protocol each time. MCP collapses that sprawl into one JSON-RPC shape that anyone can implement in a weekend. You write the server once, and every MCP-aware client — Claude Code, Claude Desktop, the various editor plugins — can use it without modification.

5.2 Why MCP Matters for GSD Specifically

Here is the part that surprises people: you don't have to tell GSD anything about your MCP servers. It figures them out on its own.

As of GSD v1.30 (see CHANGELOG entry #1603), the executor and planner sub-agents are explicitly instructed, in their system prompts, to inspect the current set of available tools before drafting a plan or executing a task, and to use any MCP tools they find as if they were first-class capabilities. That was a one-line change to the subagent prompt templates and it unlocked an enormous amount of behavior. If you connect a Postgres MCP server to Claude Code, the planner now sees `postgres.query` in its tool list and will naturally reach for it when you ask it to plan a database migration. If you connect Google Stitch's MCP, the planner sees `stitch.generate_screen_from_text` and will write plan steps that call it directly.

You don't edit a GSD config file to declare the tool. You don't hand-wire anything in `.planning/`. The subagents inherit Claude Code's entire tool surface, MCP and all, and they use what they find.

GSD's agents discover MCP tools automatically — that's what v1.30's #1603 fix added. Just configure the MCP server in Claude Code and GSD will pick it up on the next planning or execution run.

5.3 Transport Types: stdio and HTTP

MCP servers come in two flavors, distinguished by how the client talks to them.

5.3.1 stdio Transport

The most common shape is *stdio*: the client launches the server as a local subprocess and exchanges JSON-RPC messages over the child's standard input and standard output streams. This is simple, fast, and completely local — no network stack, no authentication, no ports to open. Most of the MCP servers you will install for personal use are stdio servers, invoked as `npx` or `uvx` commands. The filesystem server, the GitHub server, the official Postgres reference server, the Stitch proxy we'll meet in Chapter 7: all stdio.

The lifecycle is straightforward. When you start a Claude Code session, the client reads your MCP config, spawns each stdio server as a subprocess, speaks JSON-RPC to it over pipes, and keeps the subprocess alive for the duration of the session. When the session ends, the subprocess is killed. Restarting Claude Code restarts all your stdio servers from a clean slate.

5.3.2 HTTP Transport (Streamable HTTP)

The other shape is remote HTTP, now formally called Streamable HTTP in the MCP specification. Instead of spawning a subprocess, the client connects to an HTTP

endpoint over the network and exchanges messages as HTTP requests with server-sent events for streaming. This is how you would connect to a shared team MCP server, a cloud service that exposes an MCP endpoint, or any tool that doesn't want to run on your machine.

HTTP transport is more complicated in exchange for more power: you have to think about authentication (API keys in headers, bearer tokens, OAuth flows), endpoint availability, and the fact that your network is now in the critical path of every tool call. It is the right choice for production integrations and the wrong choice for personal utilities.

5.4 How Discovery Works

When Claude Code starts a session, it walks a short, ordered list of configuration sources and merges the results. The project-scoped file `.mcp.json` in the current working directory comes first; the user-scoped file `~/.claude.json` comes second. A server name defined in the project file takes precedence over the same name in the user file, which means you can ship a project-local override for a server that you otherwise use everywhere.

For each server entry, Claude Code either launches a subprocess (stdio) or opens a connection (HTTP), sends the MCP handshake, and asks for the server's tool listing. Those tools are added to the set of things the assistant can invoke in the current conversation. GSD's subagents, when they are spawned by commands like `/gsd:plan-phase` or `/gsd:execute-phase`, inherit the same tool surface from the parent Claude Code process. There is no second discovery step, no second config file, no second place for things to go wrong.

The upshot is that an MCP server you add once is immediately available everywhere GSD does useful work: planning, execution, verification, review. The subagents don't need to be taught about the new tool. They just find it sitting on the shelf.

5.5 A Short History and Why It Matters

MCP is young. The specification was first published in late 2024, the reference implementations appeared shortly after, and the ecosystem of third-party servers has been expanding roughly monthly since then. In practical terms this means three things. First, the protocol surface is still small enough to read in an afternoon — if you're curious about what an MCP message looks like on the wire, the spec is not yet so sprawling that you need a guide to navigate it. Second, new servers are appearing constantly, and the server you want probably exists (or will exist by next month) even if it doesn't show up on the first page of search results today. Third, the protocol is versioned and backwards-compatible in a way that means the config patterns in this guide are unlikely to break under you in the near term — when the spec evolves, it does so by adding new capabilities to the negotiation step rather than breaking old message shapes.

The reason any of this matters for a GSD user is that the protocol's youth shows

up in rough edges. Error messages from MCP servers are sometimes less polished than the equivalent error from a mature vendor SDK; documentation for less-common servers is sometimes scattered across a README, a changelog, and one Discord thread; and the first time you wire up a new server, expect to spend ten minutes figuring out which environment variable the server’s author chose to name the API key. None of this is fatal. All of it is the cost of being a year early. The payoff is that the glue code you write today will still work in three years, because the protocol itself is designed to outlive any one vendor’s implementation.

5.6 What MCP Is Not

A brief list of things MCP is not, because misconceptions in this space are expensive.

MCP is **not** a replacement for HTTP APIs. If you have a REST API that you love and that works, MCP does not make it faster or better; it wraps it. The value of MCP is not performance. It is standardization. An MCP server in front of your REST API means any MCP-aware client can use it without hand-coding the integration. That’s worth something if you have more than one client. It’s worth less if you only ever use one.

MCP is **not** an agent framework. The protocol says nothing about how the assistant decides which tool to call, how to chain tool calls together, how to handle errors, or how to plan multi-step work. That’s the assistant’s job. MCP just moves bytes between a client and a server. The intelligence lives on either side of the wire, not in the wire itself.

MCP is **not** a permissions system. Giving a tool to Claude Code via an MCP server grants the assistant whatever access the underlying credentials provide. If your Postgres MCP server connects with a superuser account, Claude Code gets superuser access to the database, and so does every subagent GSD spawns. Scope your credentials at the database, filesystem, or API level — that’s where permissions belong — and treat MCP as a transport layer that will faithfully carry whatever authority you give it.

MCP is **not** magic. The tools you install are the tools you get. If no one has written an MCP server for the thing you want to integrate with, you have two options: find a close-enough existing server (a generic HTTP fetch server, a shell-command runner, a scripting server) or write one yourself. Writing one yourself is surprisingly cheap — the reference TypeScript and Python SDKs make a minimum-viable server about fifty lines of code — but it is still work, and no amount of AI enthusiasm will conjure a server that doesn’t exist yet.

Chapter 6

Configuring MCP Servers

Now that you know what MCP is and why GSD cares, let's add one. There are three ways to register a server with Claude Code: the `claude mcp` CLI, a project-scoped `.mcp.json` file, and the user-scoped `~/.claude.json` file. All three produce the same runtime behavior; they differ only in scope and in how much typing you do.

6.1 Adding Servers via the CLI

The `claude mcp add` command is the fastest way to register a server without hand-editing JSON. Two examples cover the two transports:

```
# Register a local stdio server (the Stitch proxy, which we'll meet  
# in Chapter 7). The "--" separator marks the end of claude's own  
# flags and the start of the subprocess command line.  
claude mcp add stitch --transport stdio -- npx @_david east/stitch-mcp proxy  
  
# Register a remote HTTP server at user scope (-s user means every  
# project on this machine will see it).  
claude mcp add my-api --transport http https://api.example.com/mcp -s user
```

The CLI writes the entry to the appropriate config file for you — project or user scope, depending on the `-s` flag — and from that point on, the server is indistinguishable from one you added by editing JSON directly. Use the CLI for one-shot experimentation; use JSON files when you want to commit the config alongside the rest of your project or when you want finer control over env-var injection.

6.2 Project-Scoped Configuration: `.mcp.json`

A `.mcp.json` file in the root of your project declares servers that are specific to this project and nowhere else. The shape is dead simple:

```
{  
  "mcpServers": {  
    "filesystem": {  
      "command": "npx",
```

```
    "args": ["-y", "@modelcontextprotocol/server-filesystem",  
            "/Users/me/projects"]  
  }  
}
```

The top-level `mcpServers` object holds one entry per server, keyed by a short name of your choosing (this name becomes the prefix on tool calls — so `filesystem.read_file` would be the fully qualified tool name in the above example). The `command` is the binary to launch, and `args` is the argv passed to it. That's the minimum viable config.

6.3 User-Scoped Configuration: `~/.claude.json`

The user-scoped file lives at `~/.claude.json` and has the same shape, just in a different place. Entries here apply to every project you open in Claude Code, unless a project-local entry with the same server name overrides it. Use the user-scope file for personal utilities: your `filesystem` server pointing at `~/projects`, your GitHub server with your personal access token, whatever daily-driver search MCP you've settled on.

6.4 Project vs User Scope: How to Decide

The rule of thumb is simple. If the tool is specific to this codebase — a database server configured with this app's connection string, a Stitch project that holds this app's UI, a staging API endpoint you only ever hit from this repo — use **project scope**. If the tool is a personal utility that you'd want in every session — `filesystem`, GitHub, your search engine of choice — use **user scope**. When in doubt, start at user scope and move it down to project scope only if it turns out to need project-specific credentials or paths.

A slightly more nuanced version of the same rule: ask yourself what would happen if a teammate cloned your repo and ran Claude Code in it. Would they need this server to reproduce your work? If yes — the database MCP, the Stitch MCP pointing at a shared project, the deployment API wrapper — it belongs in project scope, committed (minus credentials) so the next developer gets it for free. If no — your personal `filesystem` server, a search MCP pointing at your own notes, a weather server you installed for fun — it belongs in user scope where it stays with you and doesn't clutter the shared config. The tiebreaker, when both feel true, is whether the tool holds state that is specific to this project. A Postgres MCP holding this app's connection string clearly does. A GitHub MCP holding your personal access token clearly does not, even if you only ever use it in one repo at a time.

There's a third scope worth knowing about, even though most users will never touch it directly: the session-local scope created by `claude mcp add` without either `-s project` or `-s user`. This adds a server for the duration of the current Claude Code session only, in-memory, with no file changes. It is useful for truly one-shot experiments — you want to poke at a new server for ten minutes without committing

to it — and it vanishes when you exit the session. Treat it as a try-before-you-buy mode.

6.5 Environment Variables in Config

Hardcoding an API key into `.mcp.json` is a loaded gun. Even if you never commit the file, it still sits on your disk in plaintext, readable by any process running as your user, and any casual `cat .mcp.json` will print it to a terminal that might be scrolled back or screen-shared or logged by an operator tool. The correct pattern is env-var interpolation: store the secret in your shell environment and reference it by name in the config.

```
{
  "mcpServers": {
    "github": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-github"],
      "env": {
        "GITHUB_PERSONAL_ACCESS_TOKEN": "${GITHUB_TOKEN}"
      }
    }
  }
}
```

The values shown above are **placeholders**. Never commit real API keys, OAuth tokens, or database connection strings to a file that will be checked into version control. Use environment variables (`${VAR_NAME}`) and add `.mcp.json` to `.gitignore` whenever it carries credentials.

The `env` block passes environment variables into the subprocess. Values of the form `${NAME}` are expanded from your own environment at the moment Claude Code spawns the server — so the real token lives in your shell profile (or a secrets manager, or a `dotenv` file that is itself `gitignored`), not in the config file. If the variable is unset, the substitution leaves an empty string and the server will almost always fail with an auth error; that’s the correct failure mode.

Add `.mcp.json` to `.gitignore` whenever it carries API keys, OAuth client secrets, or database connection strings — even if you’re using env-var interpolation, because the shape of the file itself can leak information about your infrastructure. The user-scope file `~/claude.json` lives outside any repository so it is safer for shared credentials by default, but you should still treat it like an SSH key: tight file permissions, no copies in shared drives, no pasting into chat tools. A credential that leaves your laptop is a credential that has leaked, regardless of where it ended up.

One more footgun worth flagging: some editors and IDEs offer to “helpfully” open `.mcp.json` in a JSON schema view that pings a remote schema registry. Disable this for any config file that contains internal endpoint URLs or server names you’d

rather not publish to someone else's telemetry. The GSD guide team has seen this happen once. Once is enough.

Chapter 7

Scenario --- Google Stitch + GSD

Google Stitch is an AI-powered UI generation product built on Gemini 2.5 Pro. You describe a screen in plain English (or hand it a reference image), and it produces production-ready frontend code in the framework of your choice — HTML/CSS, React, Vue, Angular, all with Tailwind variants. For developers who can ship backends in their sleep but whose last three side projects died because the UI looked like a 2003 wireframe, this is the exact tool you’ve been waiting for.

Stitch ships with an official MCP integration in the form of the `@davideast/stitch-mcp` npm package. The package provides two things: an interactive init wizard that handles the painful Google Cloud setup, and an MCP proxy server that your Claude Code client connects to. The proxy is the recommended entry point, and the rest of this scenario walks through why.

The Stitch MCP server exposes the following tools: `create_project`, `generate_screen_from_text`, `get_screen`, `get_screen_code`, `get_screen_image`, `build_site`, `list_projects`, and `list_screens`. You won’t need to invoke any of them by hand — GSD’s planner and executor subagents will find them in the tool list and wire them into the plan for you once you’ve completed setup.

Step 1 — Admit that the problem is real.

You have a meal-planning app in your head. Onboarding screen, dashboard, recipe browser, meal calendar, shopping list: five screens total. You can write the backend this weekend, and you have written it in your head three times already. What has always stopped you is that the UI from each of those mental iterations looked like a homework assignment. You don’t want to learn Figma. You don’t want to be tied to one design tool for the rest of your career. You do want to ship. This is the problem Stitch is trying to solve, and the question this chapter answers is what happens when you hand that solved problem to GSD.

Step 2 — Set up a Google Cloud project and enable the Stitch API.

Open the Google Cloud Console (console.cloud.google.com), sign in with the Google account you want to use for Stitch billing, and either create a new project or pick an existing one from the project dropdown. From the API & Services section, search the product catalog for “Stitch” and click Enable on the product page. You’ll be asked to confirm or attach a billing account; API usage bills against that account at whatever rate Google currently publishes, so check the current pricing page before you enable

it if you're cost-sensitive.

Two notes about this step that will save you a headache. First, the Stitch API is not enabled by default on any project, including new ones — you always have to click Enable explicitly. Second, an active billing account is required; the free tier is metered, not permanent, and a project without billing will fail at the first tool call with an authentication-like error that isn't actually an authentication problem. Make sure billing is attached.

Step 3 — Install stitch-mcp.

```
# Run the interactive initializer. It will walk you through
# gcloud auth, project selection, and config generation.
npx @davideast/stitch-mcp init
```

The init wizard is the part you will love. It runs `gcloud auth application-default login` for you if `gcloud` is installed and you aren't already signed in, shows you the list of projects you have Stitch access to, lets you pick one, and then prints a ready-to-paste `.mcp.json` snippet at the end. It also tells you which environment variables it expects for the API-key path if you'd rather go that route instead of OAuth.

Step 4 — Choose an auth method.

There are three ways to authenticate a Stitch MCP connection, and getting this decision right on day one is worth more than any other detail in this walkthrough. Here's the comparison:

Method	Setup	Token Lifetime	Best For
API Key	Set STITCH_API_KEY env var	Permanent	Solo devel- opers, quick local testing
OAuth (gcloud)	<code>gcloud auth application-default login</code>	1 hour, auto-refreshed by proxy	Teams, CI/CD, daily driver
Remote MCP (Google native)	<code>claude mcp add stitch --transport http</code>	1 hour, MANUAL refresh	Quick experi- ments only

The numbers in the middle column are load-bearing. Google OAuth application-default tokens have a **one-hour lifetime** — this is a Google-wide policy, not a Stitch-specific limit, and no amount of configuration will extend it. What differs between rows 2 and 3 is who handles the refresh. The local proxy from `@davideast/stitch-mcp` watches the token's expiration, re-runs the refresh flow in the background when the token gets close to expiring, and transparently keeps your session alive for as long as Claude Code is running. A manually-added remote MCP does not do this — the token expires in the middle of a working session, your next `generate_screen` call fails with a 401, and you have to manually re-authenticate before you can continue.

For anything resembling daily work, the OAuth-via-proxy path is the correct choice. The auto-refresh is the single biggest quality-of-life feature in the whole integration. The API-key path is legitimately fine for solo developers who want to get started quickly, but every committed key is a potential leak and every rotation is manual; budget for that if you go that way. The remote-MCP path is genuinely only useful for a quick experiment: spin it up, poke at one screen, tear it down. Anything longer and you'll spend more time re-authenticating than building.

Step 5 — Add MCP config to `.mcp.json`.

Paste the config block from the init wizard into a `.mcp.json` file at the root of your project. It will look like this (the proxy path, which is what you want):

```
{
  "mcpServers": {
    "stitch": {
      "command": "npx",
      "args": ["@_davideast/stitch-mcp", "proxy"]
    }
  }
}
```

Credential Safety

The values shown above are **placeholders**. Never commit real API keys, OAuth tokens, or database connection strings to a file that will be checked into version control. Use environment variables (`${VAR_NAME}`) and add `.mcp.json` to `.gitignore` whenever it carries credentials.

If you decided during Step 4 to go the API-key route instead, the config shape is slightly different — you supply the key through the `env` block rather than relying on ambient gcloud credentials:

```
{
  "mcpServers": {
    "stitch": {
      "command": "npx",
      "args": ["@_davideast/stitch-mcp", "proxy"],
      "env": {
        "STITCH_API_KEY": "${STITCH_API_KEY}"
      }
    }
  }
}
```

Credential Safety

The values shown above are **placeholders**. Never commit real API keys, OAuth tokens, or database connection strings to a file that will be checked into version control. Use environment variables (`${VAR_NAME}`) and add `.mcp.json` to `.gitignore` whenever it carries credentials.

Reminder from Chapter 6: any `.mcp.json` that carries credentials — API keys in `env`, connection strings, bearer tokens — belongs in `.gitignore`. The `env-var` interpolation

pattern (`${STITCH_API_KEY}`) pushes the actual secret out of the committed file and into your shell environment, which is strictly better than a literal, but doesn't replace gitignoring the file entirely for credential-bearing configs.

Step 6 — `/gsd:new-project`.

Now the fun part. Fire up Claude Code in your empty project directory and run:

```
/gsd:new-project a meal-planning app with onboarding, dashboard,
  recipe browser, meal calendar, and shopping list --- 5 screens
  total. Use Stitch for UI generation with React and Tailwind.
```

GSD's `new-project` command writes `.planning/PROJECT.md` with a one-paragraph description, a rough feature list, and a detected tech stack. This is also the moment it kicks off the *discuss* phase, which is the structured conversation where design decisions get locked in before any code is written. At this point GSD does not yet know it will be using Stitch — it has inferred "React" and "Tailwind" from your prompt, but it hasn't yet seen the Stitch tools in its tool list because the discuss phase hasn't run. That's fine. The next step is where the MCP layer starts to earn its keep.

Step 7 — `/gsd:discuss-phase 1`.

The discuss phase is where you lock in decisions that the planner will treat as constraints. This is where you commit to React + Tailwind as the framework (so the planner doesn't hedge later), where you decide whether to let Stitch pick a color palette or supply one yourself, and where you give the planner the Stitch project ID it will need to call the MCP tools. Here's a fragment of what you might see in the resulting `DISCUSS.md`:

```
## Design Decisions --- Phase 1

Framework: React 18 + Tailwind CSS 3
UI tool: Google Stitch (via stitch-mcp proxy)
Stitch project ID: YOUR_PROJECT_ID
Color palette: defer to Stitch (will regenerate if needed)
Target screens: onboarding, dashboard, recipes, calendar, shopping

Rationale: Stitch generates React + Tailwind natively, so no
transpilation step. Letting Stitch pick the palette on the first
pass gives us a starting aesthetic to react to --- easier than
designing from a blank page.
```

The Stitch project ID is the key piece. Without it the planner can reach the Stitch MCP's tool list but won't know which project to operate in; with it, every subsequent tool call will target the correct project automatically. Grab the ID from Stitch's project listing (or from `stitch.list_projects` if you'd rather ask the MCP server itself) and paste it into the discuss phase output.

Step 8 — `/gsd:plan-phase 1`.

With the discuss phase committed, run `plan-phase`. The planner subagent spawns with access to Claude Code's full tool surface — which now includes every tool the Stitch MCP exposes. It reads `DISCUSS.md`, sees that Stitch is the chosen UI tool, sees `stitch.generate_screen_from_text` sitting in its tool list, and writes plan steps that call it directly.

Here's the flavor of what the resulting `PLAN.md` will look like once the planner is done:

```
Task 3: Generate onboarding screen via Stitch
Tool:  stitch.generate_screen_from_text
Inputs: {
  project_id: YOUR_PROJECT_ID,
  prompt:    "Onboarding screen for a meal planning app. Three
             steps: welcome, dietary preferences, household size.
             Warm, approachable tone. Tailwind utility classes.",
  framework: "react-tailwind"
}
Output: HTML/CSS in src/screens/Onboarding.tsx
Depends on: Task 1 (project scaffold), Task 2 (stitch auth check)
```

The planner generates one of these blocks per screen — five screens, five Stitch calls — plus the usual scaffolding, routing, and integration tasks. Notice that the planner is writing tool-specific inputs with real Stitch-compatible field names (`project_id`, `framework`, `prompt`). It knows to do that because the MCP tool's schema tells it what arguments the tool accepts. Schemas beat prose every time.

Step 9 — `/gsd:execute-phase 1` and `/gsd:ui-review 1`.

Execute-phase is where the rubber meets the road. The executor subagent walks through the plan task by task, calling MCP tools for the tasks that declare them, and writing real files on disk as the tools return. You'll see progress output along the lines of:

```
[Task 3/14] Generate onboarding screen via Stitch
-> Calling stitch.generate_screen_from_text
-> Received screen_id = scr_a8f20b1e
-> Calling stitch.get_screen_code (framework=react-tailwind)
-> Wrote src/screens/Onboarding.tsx (2.4 KB)
-> Calling stitch.get_screen_image (for reference asset)
-> Wrote docs/screens/onboarding.png (47 KB)
[Task 3/14] COMPLETE
```

Each screen produces a real file in `src/screens/` and a reference PNG in `docs/screens/`. The reference PNGs are not strictly necessary but they're worth keeping: they're what Stitch's renderer produced for the same prompt, and later on they make it much easier to reason about visual drift when you iterate on the copy or the palette.

Once the executor has run through all five screens, run `/gsd:ui-review 1`. The ui-review command audits the generated UI against GSD's six-pillar UI standards: accessibility, usability, consistency, information density, responsive behavior, and visual polish. It will flag missing alt text, contrast failures, keyboard traps, and the other quiet ways AI-generated UI can be technically correct and practically broken. Stitch's output is generally good, but it is not infallible, and ui-review is where you find out.

Credential Safety

The values shown above are **placeholders**. Never commit real API keys, OAuth tokens, or database connection strings to a file that will be checked into version control. Use environment variables (`${VAR_NAME}`) and add `.mcp.json` to `.gitignore` whenever it carries credentials.

Stitch can generate in at least six frameworks (HTML/CSS, React, Vue, Angular, with or without Tailwind). Tell GSD which framework you want during the *discuss* phase, not the *execute* phase. The planner will then write framework-specific task inputs that the executor can hand directly to `stitch.get_screen_code` without a conversion step. Waiting until execute-time means the planner has already committed to a structure it now has to rewrite.

Chapter 8

Scenario --- Database MCP + GSD Planning

GSD’s planner is smart but not clairvoyant. Ask it to “add a `user_preferences` table” in a project where it has no visibility into the database, and it will produce a plausible migration against a plausible schema — not your schema. The columns it picks will be reasonable guesses, the foreign key names will look like what a textbook would suggest, and the migration will fail the moment it meets the real database. Worse, a junior reviewer glancing at the plan might not notice.

A Postgres MCP server fixes this by giving the planner read-only introspection into the actual schema. With the MCP installed, the planner issues real `SELECT` statements against `information_schema` before it drafts the migration, and the resulting plan references the columns that actually exist. The rest of this chapter walks through the setup.

Step 1 — Install a Postgres MCP server.

The Anthropic-maintained reference server `@modelcontextprotocol/server-postgres` is the safest starting point. It is deliberately **read-only**: the only tool it exposes is a query function that runs against the database, and the server wraps every query in a read-only transaction so even a `DELETE` inside the query text cannot mutate data. That’s a feature, not a limitation. For planning work, read-only is all you need and all you want.

```
# No install step is needed -- npx fetches the package on demand
# when Claude Code launches the subprocess. But you can pre-cache
# it to shave a few seconds off the first session:
npx -y @modelcontextprotocol/server-postgres --version
```

Step 2 — Configure with a connection string.

Add the server to your `.mcp.json` with a connection string supplied as an environment variable. Never hardcode the string into the file — database URLs contain credentials by definition, and a credential in a config file is a credential one `git add` away from leaving the building.

```
{
  "mcpServers": {
    "postgres": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-postgres"],
      "env": {
        "DATABASE_URL": "${DATABASE_URL}"
      }
    }
  }
}
```

Credential Safety

The values shown above are **placeholders**. Never commit real API keys, OAuth tokens, or database connection strings to a file that will be checked into version control. Use environment variables (`${VAR_NAME}`) and add `.mcp.json` to `.gitignore` whenever it carries credentials.

Export `DATABASE_URL` in your shell before starting Claude Code (from a `dotenv` file, your secrets manager, or a line in your shell profile — whichever you prefer). The substitution happens when Claude Code spawns the MCP subprocess, so the value needs to be set in the shell that launches Claude Code, not just in Claude Code's own session. And yes: this `.mcp.json` belongs in `.gitignore` for exactly the reasons Chapter 6 covered.

Step 3 — /gsd:plan-phase with schema visibility.

With the Postgres MCP running, start a plan-phase command that will require schema knowledge. For example:

```
/gsd:plan-phase 2 add a user_preferences table with notification
  toggles, theme selection, and default meal calendar view
```

The planner subagent sees `postgres.query` in its tool list and does the right thing without being told: before drafting any migration steps, it inspects the existing schema with queries like:

```
SELECT column_name, data_type
FROM information_schema.columns
WHERE table_name = 'users'
ORDER BY ordinal_position;
```

It does this for every table the new migration might touch, not just the obvious ones, because it was trained to build context before acting. The returned rows get folded into the planning context, and the migration steps the planner writes are grounded in what actually exists.

Step 4 — Observe PLAN.md referencing real columns.

The resulting plan will include a block that looks like this:

```
Task 5: Migration -- create user_preferences table
  Preconditions verified via postgres.query:
  - users table exists with columns (id, email, created_at,
    updated_at, last_login_at)
```

```
- no existing user_preferences table
Migration SQL:
ALTER TABLE users ADD COLUMN preferences JSONB;
-- or, if we prefer a separate table:
CREATE TABLE user_preferences (
  user_id INTEGER PRIMARY KEY REFERENCES users(id),
  ...
);
```

That comment block listing preconditions is the key output of the whole exercise. A human reviewer can look at the plan and verify, at a glance, that the planner saw the real schema when drafting the migration. The reviewer’s job is no longer “spot-check every column name against the real database” — it is the much smaller job of “confirm the preconditions the planner listed are the right ones.”

Step 5 — Cleanup.

There is no cleanup. The Postgres MCP server is read-only and stateless: it holds no data on disk, writes no files, and makes no changes to your database. Killing Claude Code ends the subprocess; starting a new session respawns it from scratch with the same config. The only persistent trace is whatever lives in your query logs, which is a database-side concern and not a GSD one.

The Postgres MCP server is read-only. If you want the generated migrations to actually run, that’s a separate concern — GSD plans them, you (or your CI) execute them through your normal migration tool (Flyway, Liquibase, Alembic, sqlx-migrate, whatever you already use). Do not look for an MCP server that will run migrations for you. The separation of planning from execution here is not a limitation; it is the safety boundary that keeps a planning error from becoming a production outage.

Chapter 9

Building MCP-Aware GSD Workflows

The two scenarios in Chapters 7 and 8 each used one MCP server in isolation. Real projects accumulate servers the way shell profiles accumulate aliases — a filesystem server, a GitHub server, a Postgres server for the main database, maybe a separate analytics Postgres, Stitch for UI, a custom server that wraps your internal deployment API. This chapter is about making the whole pile work together, and about the edges where MCP integration gets interesting.

9.1 Combining Agent Skills with MCP

Part I of this guide introduced agent skills — the small project-scoped prompt fragments that get injected into a subagent’s context whenever it runs in this project. Skills and MCP tools compose in a way that is more powerful than either one alone.

The trick is that a skill can refer to an MCP tool by name. A skill is just prose injected into the planner’s or executor’s prompt, and if that prose says “when planning database migrations, always call `postgres.query` first to inspect existing tables,” the planner will do exactly that. The skill doesn’t need to know how `postgres.query` works; it just needs to mention the name. The MCP server already told Claude Code about the tool’s schema during discovery, so the planner can fill in the arguments correctly on its own.

This pattern is how you bake team conventions into a project. Your team decides that every new feature needs a database schema check before planning? Write a one-paragraph skill that says so, mention the `postgres.query` tool by name, and every subsequent `/gsd:plan`-phase run will follow the rule without anyone having to remember to invoke it. The same trick works for any MCP server: a skill that says “for any UI task, always call `stitch.list_projects` first to confirm the Stitch project exists” costs you a paragraph and saves you a class of “undefined project” errors you would otherwise hit at execute time.

A worked example makes this concrete. Suppose your team has agreed that every database-touching phase must start by reading the current schema, and every UI-touching phase must start by confirming the Stitch project exists and has the

expected screen count. You can bake both rules into a skill like this (placed in `.claude/skills/team-conventions.md` or wherever your project's agent skills live):

```
# Team conventions for planning and execution

## Before any database migration task
1. Call postgres.query with a schema inspection query
   against information_schema.tables and information_schema.columns
   for the affected tables.
2. Include the returned column list in the task preconditions.
3. Only then draft the migration SQL.

## Before any Stitch-generated UI task
1. Call stitch.list_projects and confirm the configured project
   ID is present.
2. Call stitch.list_screens for that project and record the
   current screen count.
3. If the screen count differs from what DISCUSS.md claims, stop
   and surface the mismatch instead of generating blindly.
```

The skill is maybe thirty lines of prose. The behavior change is substantial: you stop finding out about schema drift at execute time and start finding out about it at plan time, when fixing it is cheap. Notice that this pattern does not require any changes to GSD's source code, to the MCP servers themselves, or to Claude Code's configuration. It is entirely text, and the text flows through the same discovery-and-injection paths you already use for every other skill.

The reason this works is worth pausing on. Subagent prompts are assembled fresh at the start of each subagent invocation: the base template, the current phase's context, any applicable skills, and the current tool manifest. When a skill mentions a tool name, the planner sees that mention *and* the tool's schema side by side. It doesn't have to guess what arguments the tool takes, because the schema is right there. It doesn't have to be reminded to use the tool, because the skill says so in plain language. Skills and MCP compose cleanly because they're both contributing to the same assembled prompt from different directions.

9.2 The GSD Plugin Format (Issue #1883)

There's a larger integration idea in the pipeline. GSD issue #1883 tracks a plan to package GSD itself as a Claude Code plugin — a single installable unit that would bundle GSD's skills, subagent templates, and a dedicated MCP server, all in one place. The promise is that a user could run one install command and get the entire GSD experience, with its own MCP surface exposed as a first-class tool set, rather than having GSD live alongside MCP as a separate concern.

This is on the roadmap for v1.32 or later — the plugin format is still evolving, and the schema the plugin manifest will use is not yet fixed. Committing to a specific shape in this guide would be premature. The thing to know is that it's coming, that it is specifically designed to make MCP integration a first-class part of GSD rather than an afterthought, and that the scenarios in this part will still work on the plugin

version because the underlying primitives are unchanged. Follow the issue if you want early visibility.

9.3 Failure Modes: When an MCP Server is Unavailable

An MCP server can fail mid-execution. The subprocess crashes, the remote endpoint returns a 500, the network is flaky, the auth token expires at exactly the wrong moment. What happens then?

Claude Code returns a tool error to the agent that made the call. The error is not silent: it propagates back as a failed tool response with an error message attached. GSD’s executor subagent sees the error and does one of two things.

The first option is a retry, chosen when the failure looks transient — a network timeout, a 503, an obviously flaky response. The executor will retry the same tool call, sometimes with a short delay, before concluding that the problem is real. This handles most of the everyday flakiness you encounter with remote servers.

The second option is to mark the task as blocked and write a deviation note to the execution log. A deviation note is the executor’s way of saying “I can’t finish this task and the reason is not my fault.” It includes the original plan step, the tool error, and enough context for a human to resume work later. The phase does not silently complete; it ends in a blocked state, and the status command (`/gsd:status` or similar) will surface the blockage the next time you ask.

The plan does not degrade silently. Failed MCP calls are visible in the executor log, deviation notes make them unmissable, and the phase-completion gate refuses to pass a phase where execution was blocked on a tool error. If you’ve been running GSD long enough to trust it, this is the behavior you want — the tool doesn’t lie about what happened, it tells you something broke and stops.

9.4 Observability: Knowing What Your MCP Servers Did

An underappreciated part of running MCP at scale is knowing what your servers have actually been doing. Claude Code’s conversation log shows you every tool call the assistant made, but once you have five or six MCP servers running, that log can get noisy, and correlating “this file on disk” with “this tool call that produced it” is not always easy from the transcript alone.

Three patterns help. First, most MCP servers can be started in a verbose or debug mode that prints each tool invocation to `stderr`. Claude Code surfaces server `stderr` in its own logs, so turning on verbose mode on a suspect server is often the fastest way to catch a misbehaving call. The flag is server-specific (look for `--verbose`, `--log-level debug`, or an environment variable like `LOG_LEVEL=debug`), but it exists almost everywhere because MCP server authors are themselves debugging their own servers against Claude Code and need the same signal.

Second, for servers you write or control, emit a one-line audit record per tool call: timestamp, tool name, argument summary, outcome, duration. Pipe it to a file or to

your logging pipeline. When something goes sideways at execute time, the audit log tells you what actually happened versus what the plan says was supposed to happen — and those two narratives are not always the same.

Third, for production-grade setups, consider wrapping your MCP servers behind a single reverse-proxy MCP that logs all traffic and then forwards calls to the appropriate underlying server. This adds a hop but gives you one place to look for everything. It's overkill for personal projects and exactly right for shared team infrastructure where “who called what, when, against which database” is a compliance question with an actual answer.

9.5 Version Drift: What Happens When a Server Changes

Your MCP servers will change under you. The Stitch proxy will publish a new version that renames a tool argument. The Postgres server will add a new capability you've been wanting. A third-party server you depend on will be deprecated in favor of a successor that is almost but not quite backwards-compatible. None of this is exotic; it's normal package ecosystem behavior, and MCP is no different from npm or pip in that regard.

GSD copes with this better than you might expect, because the discovery step happens at the start of every session. The planner sees the current set of tools with the current schemas — not a cached snapshot from when you first configured the server. If a tool's argument list changes overnight, the next plan-phase run will use the new shape without you having to update anything in GSD. If a tool disappears entirely, the planner will stop proposing it, because it won't see it.

The failure mode to watch for is a skill that names a tool explicitly. If your team-conventions skill says “always call `postgres.query`” and someone upgrades the Postgres MCP to a version where that tool was renamed to `postgres.execute`, the skill will keep asking for the old name, the planner will try to call it, and the call will fail. The fix is to update the skill to match. A short CI check that grep-s your skills for known MCP tool names and cross-references them against a list of currently-installed servers is cheap and will save you a debug session.

A short checklist of what you should take away from Part II:

- **Project-scoped servers** for project-specific tools: the database for this app, the Stitch project for this UI, the staging endpoints for this service.
- **User-scoped servers** for personal utilities: filesystem, GitHub, your daily-driver search MCP, anything you'd want available in every Claude Code session.
- **Env-var your credentials** every time, without exception. Never paste a real API key into a committed file, and never paste one into a file you're “about to” gitignore.
- **.mcp.json belongs in .gitignore** any time it carries secrets, even if you're using env-var interpolation. The shape of the file itself can leak information you don't want to publish.

- **For teams, prefer OAuth with service accounts** over shared API keys. Per-user auth with proper rotation is the only approach that survives a team member leaving. Shared keys are a time bomb with a known fuse length.
- **Token expiration is real:** OAuth tokens die after an hour. Use proxies that auto-refresh. Do not use raw remote MCP auth for anything you plan to run longer than a demo.

The overarching theme of Part II is that MCP is plumbing, and good plumbing disappears. Once your servers are configured, the gitignore is in place, and the env vars are exported, you stop thinking about MCP at all. GSD’s planner writes plans that use the tools you’ve plugged in, the executor calls them, and the outputs land on disk. The configuration work is upfront and small. The payoff is that every subsequent plan and every subsequent execution has access to the real state of your project — the real schema, the real Stitch project, the real repository — and the quality of the resulting work rises accordingly. That is the entire argument for connecting GSD to MCP, and it is the reason Part III will assume all of this is in place when it turns to multi-repo and monorepo workflows.

Part III

Multi-Repo Workflows

Chapter 10

Workstreams vs. Workspaces --- When to Use Which

Parts I and II taught GSD to speak your project’s dialect and to reach outside of itself. Part III is about a more mundane but equally painful problem: real projects are rarely a single repository with a single timeline. Even when a project *is* a single repo, it often runs several independent tracks of work in parallel — a backend feature, a frontend redesign, an infrastructure migration, a one-off experiment — each with its own planning cadence, its own owners, and its own definition of “done”. And when the project genuinely spans multiple repositories, as microservice fleets and polyrepo libraries often do, the coordination problem multiplies.

GSD has two different answers to that problem, and learning when to reach for each is the difference between a smooth multi-track workflow and a pile of overwritten STATE.md files.

10.1 The Problem: Shared Planning State

The default assumption baked into GSD is that a repository has one active plan at a time. The files in `.planning/` — STATE.md, ROADMAP.md, REQUIREMENTS.md, the phase directories — describe a single timeline of intent. Commands like `/gsd:progress` and `/gsd:next` read that timeline to decide what to tell you. `/gsd:execute-phase` writes to it.

If you try to run two unrelated workstreams against that shared state at the same time, bad things happen. STATE.md flips between them as each track edits it. The phase numbering collides. `/gsd:progress` reports “phase 4 in progress” when you meant that for the backend feature, and the frontend lead reads it and wonders why their redesign lost two phases overnight. The ROADMAP gets clobbered because the last writer wins. Every single one of those symptoms traces back to the same root cause: two independent timelines trying to share one set of planning files.

Workstreams and workspaces both solve that root cause. They solve it at different layers of the stack, and they have different costs.

10.2 Workstreams: One Repo, Many Timelines

Workstreams were introduced in GSD v1.28. The idea is dead simple: *if you already have one repo, keep one repo, but give each independent timeline its own isolated planning subtree*. Under the hood, a workstream is a directory under `.planning/workstreams/` that contains its own `STATE.md`, `ROADMAP.md`, `REQUIREMENTS.md`, and phase directories — a complete parallel planning tree, scoped under a name of your choice. When you switch to a workstream, GSD transparently re-roots all its planning operations to read and write from that subtree. The root `.planning/` is untouched.

The commands are short and predictable. `/gsd:workstreams create <name>` creates a new one. `/gsd:workstreams switch <name>` makes it the active workstream for subsequent commands. `/gsd:workstreams list` shows all of them. `/gsd:workstreams complete <name>` archives one when its feature ships. The switch itself is essentially free — GSD just updates a pointer to the active workstream directory — so you can flip between two or three of them many times a day without thinking about the cost.

What workstreams *don't* do is give you git isolation. The code is still the same checkout. If two workstreams both want to edit `src/billing.ts` on the same branch, they will conflict in the same way they would without workstreams. Workstreams isolate *planning state*, not *source state*. For many real projects that distinction is exactly right, because the planning collision is the painful one and the source collision is already handled by git.

10.3 Workspaces: Many Repos, One Plan

Workspaces are the heavier instrument, also introduced in v1.28. A workspace is a separate directory — by convention under `~/gsd-workspaces/<name>/` — that contains one or more named repositories, each checked out as either a git worktree or a full clone, plus a workspace-level manifest file called `WORKSPACE.md`. Each repo inside the workspace has its own independent `.planning/` subtree. The workspace itself is the coordination layer: it knows which repos are in the workspace, tracks cross-repo status in its manifest, and lets you run coordinated planning without having to merge the repos into a monorepo you didn't want.

The commands mirror the workstream ones but operate at the repo-set level. `/gsd:new-workspace --name v2-launch --repos api,web-frontend,admin-dashboard` creates a workspace named `v2-launch` containing three repos. `/gsd:list-workspaces` shows all configured workspaces. `/gsd:remove-workspace v2-launch` tears one down when it's done. The `--strategy` flag takes one of two values:

- `worktree` (the default) creates a git worktree of each named repo. Worktrees share object storage with the parent checkout, so they're cheap to create and cheap to keep around. The trade-off is that you can't have two worktrees on the same branch of the same repo at the same time — git refuses.
- `clone` performs a full `git clone` of each named repo into the workspace direc-

tory. This is slower and uses more disk, but it gives you complete isolation — the workspace’s copy of each repo is genuinely independent from your main checkout and can sit on any branch without conflict.

Workspaces cost more to create than workstreams because they involve actual git operations — worktree creation or cloning — and a new directory tree on disk. A workspace with three repos takes a few seconds to materialize, not the sub-millisecond switch of a workstream. That’s the right cost when you actually need repo-level isolation, and the wrong cost when you don’t.

10.4 Decision Table

The shortest path to the right answer is to ask: *do my parallel timelines share code, or do they share planning?* If they share code, a workstream is what you want. If they need different code or different git state, a workspace is what you want.

Scenario	Use	Why
Backend + frontend in same repo	Workstream	Same code, different planning timelines. The extra cost of a workspace buys you nothing.
Three microservices needing coordinated planning	Workspace (multi-repo)	Different repos, each already has its own .planning/. The workspace is the coordination layer.
Feature branch isolation, current repo only	Workspace --repos .	A single-repo workspace gives you a worktree of the current repo with its own independent planning state.
Disposable spike or prototype	Workspace --strategy clone	Full isolation. When the spike dies, remove the workspace and nothing leaks back into the main checkout.
Refactor touching several repos atomically	Workspace --strategy worktree	Coordinated planning across the set, shared git fast-path under the hood.

Workstreams are much cheaper than workspaces. If the work you're about to start fits in a single repo, default to creating a workstream and only escalate to a workspace if you genuinely need git-level isolation. It is always easier to graduate a workstream into a workspace later than it is to undo a premature workspace.

10.5 One More Consideration: How Long Does the Work Live?

A secondary question worth asking before you pick is how long the timeline is expected to live. Workstreams are well suited to long-lived, repeating tracks: “the backend team’s current feature”, “the ongoing observability migration”, “this quarter’s performance work”. They rotate through many features over their lifetime, and the fact that their STATE.md persists across those features is useful.

Workspaces are better suited to *cohorts of work that share a release* — the kind of thing where you’d otherwise say “we’re shipping v2 across three services and I need to track all three together”. When the cohort ships, you remove the workspace. The repos themselves live on, but the workspace’s WORKSPACE.md and its coordination state get archived or deleted along with the launch it tracked.

Neither pattern is wrong for the other’s use case, but using a workstream for a launch cohort tends to feel fiddly (you end up making a workstream in each repo and hand-coordinating) and using a workspace for a long-lived single-repo track tends to feel heavy (you end up with a workspace directory that never quite gets cleaned up). Match the lifetime of the construct to the lifetime of the work.

Chapter 11

Scenario --- Monorepo with Independent Apps

You run a single git repository containing three apps under `apps/`: a Node backend API, a Next.js frontend, and a React Native mobile app. The three teams work on different cadences — the backend team is mid-migration to a new ORM, the frontend team is redesigning the onboarding flow, the mobile team is fighting a long-tail of platform-specific bugs. They all share the same git history and a fair amount of code under `packages/`, so splitting into three repositories would be more disruptive than it is worth. But running all three teams' GSD plans against the same root `.planning/` has already produced one `STATE.md` corruption this month, and you are done with that.

This scenario walks through giving each team its own workstream, running an independent GSD flow in each, and archiving them cleanly when features ship. Total elapsed time: about five minutes of commands, most of which are the normal GSD flow you already know.

Step 1. Starting layout.

The repository looks like this before you touch anything:

```
my-monorepo/
|-- apps/
|   |-- backend/
|   |-- frontend/
|   `-- mobile/
`-- .planning/
    |-- workstreams/
    |   |-- backend-api/
    |   |   |-- ROADMAP.md
    |   |   `-- STATE.md
    |   `-- frontend-redesign/
    |       |-- ROADMAP.md
    |       `-- STATE.md
    `-- config.json      <- shared config across workstreams
```

(The `.planning/workstreams/` subtree is what we're about to create. The root

.planning/config.json already exists and stays where it is — it holds the shared config all workstreams inherit from.)

Step 2. Create the backend workstream.

```
| /gsd:workstreams create backend-api
```

GSD creates .planning/workstreams/backend-api/ and scaffolds a fresh set of planning files inside it — an empty ROADMAP.md, an initial STATE.md pointing at an un-planned phase, and a small WORKSTREAM.md marker file that the tooling uses to detect that the subtree is a workstream rather than a plain directory. The root .planning/ is not touched. You should see output along the lines of:

```
| Created workstream: backend-api
| Path: .planning/workstreams/backend-api/
| Status: active
```

Step 3. Switch to the new workstream.

```
| /gsd:workstreams switch backend-api
```

Switching makes backend-api the active workstream. Every subsequent /gsd:* command in this session will read and write state from .planning/workstreams/backend-api/ instead of the root .planning/. There's no second-command mystery here: if you forget to switch, /gsd:progress will tell you which workstream it's reporting on in its first line, so you catch the mistake early.

Step 4. Run a normal GSD flow, scoped to backend.

At this point, everything looks and feels like vanilla GSD. You start a new project for the ORM migration:

```
| /gsd:new-project
| /gsd:discuss-phase 1
| /gsd:plan-phase 1
| /gsd:execute-phase 1
```

The PLAN.md, STATE.md, and phase directories that come out of this flow all live under .planning/workstreams/backend-api/. The root .planning/ does not see any of it. If you run /gsd:progress right now, you'll get the backend workstream's view — phase 1, in progress, whatever your executor is chewing on.

Step 5. Create the frontend workstream in parallel.

While the backend team keeps going, the frontend lead wants to start the redesign plan. In a new terminal (or just later in the same terminal), they run:

```
| /gsd:workstreams create frontend-redesign
| /gsd:workstreams switch frontend-redesign
| /gsd:new-project
| /gsd:discuss-phase 1
```

This creates a second, entirely independent planning subtree at .planning/workstreams/frontend-redesign/. The two workstreams don't see each other's state files. They can be in completely different phases, have completely different agent configurations, use different project_code prefixes — the isolation is at the planning-state level, not at the git level.

Step 6. List workstreams.

When you want to see what's going on across the whole repo, ask:

```
| /gsd:workstreams list
```

The output is a small table, one row per workstream, giving you name, status, current phase, and when it was last touched:

```
| NAME           STATUS   PHASE   LAST TOUCHED
| backend-api    active  1       3 minutes ago
| frontend-redesign active  1       1 minute ago
```

This is the one command that crosses workstream boundaries. Everything else respects the active-workstream scope and only shows you one workstream’s view at a time.

Step 7. Complete and archive.

Two weeks later, the backend ORM migration ships. You finish the last phase, run your tests, push the merge, and then retire the workstream:

```
| /gsd:workstreams complete backend-api
```

Completion does not delete anything. GSD moves the workstream’s directory to `.planning/workstreams/archive/backend-api/`, sets its status to archived, and stops surfacing it in the default `list` output. If you ever need to resurrect it — say to investigate a post-ship question — you can switch back to the archived workstream and treat it as read-only evidence.

Step 8. Cleanup.

Repeat the `complete` step for the frontend workstream when its redesign ships. The root `.planning/` has stayed clean throughout: all the actual work lived under `.planning/workstreams/`, and all the archived evidence lives under `.planning/workstreams/archive/`. Your repo is ready for the next set of parallel tracks, and there’s a searchable record of what the old tracks did.

Each workstream’s `/gsd:progress` only reports on that workstream’s state. If you want a cross-workstream view, run `/gsd:workstreams list`. This is the right scoping default — a backend dev should see backend state by default, not be distracted by frontend state — but it does mean that “all-workstreams” views are an explicit command rather than the default.

Chapter 12

Scenario --- Microservice Fleet

You have three separate repositories on disk: `api`, `web-frontend`, and `admin-dashboard`. They're independent services with independent deployment pipelines, but they all ship together as part of the v2 launch, and the product manager would like a single view of "how is v2 going?" that doesn't require opening three different planning files in three different checkouts. A workspace is exactly the right shape for this: it gives you a coordination layer on top of the three repos without forcing you to merge them into a monorepo.

This scenario walks through creating the workspace, running coordinated planning across all three repos, and tearing the workspace down cleanly when v2 ships.

Step 1. Create the workspace.

```
/gsd:new-workspace --name v2-launch \  
  --repos api,web-frontend,admin-dashboard
```

GSD looks up each named repo — either via relative paths, or via registered repo aliases (you can list the registered aliases with `/gsd:list-workspaces`'s config output) — and materializes the workspace directory under `~/gsd-workspaces/v2-launch/`. For each repo, GSD creates a git worktree (the default strategy) pointing at the repo's current branch and drops it inside the workspace. It also writes a `WORKSPACE.md` manifest at the workspace root listing the repos and their current phase status.

The whole operation takes a few seconds. At the end of it, you have a fully usable workspace containing three independently planned repositories, sitting side by side.

Step 2. Inspect the workspace tree.

```
~/gsd-workspaces/v2-launch/  
|-- WORKSPACE.md          <- workspace manifest (orchestrator-owned)  
|-- api/                  <- repo 1 (independent .planning/)  
|   |-- .planning/  
|   |   |-- ROADMAP.md  
|   |   `-- STATE.md  
|   `-- src/  
|-- web-frontend/        <- repo 2 (independent .planning/)  
|   |-- .planning/  
|   `-- src/  
`-- admin-dashboard/     <- repo 3 (independent .planning/)
```

```
|-- .planning/
|-- src/
```

WORKSPACE.md is the orchestrator-owned manifest — it tracks which repos belong to the workspace and what the cross-repo phase status is. Each repo below it is an independent checkout with its own .planning/ subtree. The three .planning/ trees are genuinely independent: they don't share state, they don't share phase numbers, they don't share REQUIREMENTS.md. The WORKSPACE.md manifest is the only thing that knows about all three at once.

Step 3. cd into each repo and run its own GSD flow.

Inside the workspace, each repo behaves like a normal GSD project. You change into one repo, run the normal commands, and the normal files land in that repo's .planning/:

```
cd ~/gsd-workspaces/v2-launch/api
/gsd:new-project
/gsd:discuss-phase 1
/gsd:plan-phase 1
```

Then the same in web-frontend and admin-dashboard, each with its own plan. There is deliberately no shared planning file across the three, because the teams work in different languages, different release cadences, and different levels of detail. What they share is the workspace manifest, which is updated by the orchestrator — never by the individual executors — as each repo's phase status changes.

Step 4. Execute phases in each repo.

Phase work happens locally in each repo. When you run /gsd:execute-phase inside api, the executor operates on api's files and writes to api's .planning/STATE.md. When the phase finishes, the orchestrator reads the final state and updates WORKSPACE.md at the workspace root so that the cross-repo view stays accurate. If you do nothing, the manifest will drift behind; if you run the normal GSD flow, it stays current without any extra commands from you.

Step 5. Strategy comparison: worktree vs. clone.

The --strategy flag at workspace-creation time is worth understanding, because the two strategies have meaningfully different cost profiles:

Strategy	Speed	Disk	Isolation	Best for
worktree	Fast (seconds)	Minimal (shared projects)	Shared git history	Coordinated multi-repo planning; most launch cohorts
clone	Slower (full clone per repo)	Full working copy each	Complete	Spikes you may throw away; same-branch parallel work

The worktree strategy is the default because it's almost always what you want. It shares git object storage with the parent checkout on disk, so materializing three worktrees of large repos is essentially free. The one constraint to be aware of is that git refuses to have two worktrees of the same repo on the same branch at the same time; if the workspace and your main checkout both want to live on main, the workspace's worktree has to be on a different branch.

Use the clone strategy when you specifically need full isolation — for example, when you're running a disposable experiment and don't want git objects from the spike leaking back into your main checkout, or when you need the workspace and main

checkout to sit on exactly the same branch at the same time.

Step 6. List and inspect.

When you want to see all your workspaces, or pull up the manifest for one:

```
/gsd:list-workspaces  
cat ~/gsd-workspaces/v2-launch/WORKSPACE.md
```

`/gsd:list-workspaces` prints a table of every workspace on the machine, showing name, repos, strategy, and last activity. `WORKSPACE.md` is a plain markdown file, human-readable, that captures the orchestrator’s cross-repo view — phase status per repo, outstanding blockers, last update timestamp. You can skim it in any editor without running GSD.

Step 7. Cleanup.

When v2 finally ships, you retire the workspace:

```
/gsd:remove-workspace v2-launch
```

This command asks you to confirm the workspace name before proceeding — a small guardrail against fat-fingering the wrong name and destroying an active workspace. Once confirmed, GSD removes the workspace directory, tears down the worktrees (git reports each one as pruned), and cleans up the `WORKSPACE.md` manifest. The underlying repos themselves are untouched: the `api`, `web-frontend`, and `admin-dashboard` checkouts elsewhere on your disk are completely unaffected.

Workspaces refuse removal with uncommitted changes

`/gsd:remove-workspace` will **refuse** to remove the workspace if any constituent repo has uncommitted changes. The protection is intentional: uncommitted work in a worktree is easily lost if the worktree is removed before being committed back, and “I just meant to clean up” is exactly how people lose an afternoon’s work. Either commit, stash, or push your work in each repo first, then re-run the removal. If you are absolutely sure you want to discard the uncommitted state, do it explicitly — stash or reset each repo by hand — so that the decision is deliberate.

Part II introduced project-scoped `.mcp.json` as the right way to configure an MCP server. That pattern composes cleanly with workspaces: a Stitch MCP server configured inside `~/gsd-workspaces/v2-launch/web-frontend/.mcp.json` lives inside that workspace’s copy of the `web-frontend` checkout and does not bleed into a separate workspace’s `web-frontend` checkout, or into your main `web-frontend` checkout elsewhere on disk. Multi-repo isolation and MCP scoping compose exactly the way you’d expect them to.

Chapter 13

Worktree Isolation Deep Dive

This chapter is the most technically delicate in Part III. The details matter, and one of them in particular — the STATE.md ownership rule — is the kind of thing that will silently corrupt your planning files if you get it wrong. Read carefully.

13.1 What `workflow.use_worktrees: true` Does

In `.planning/config.json`, a workflow setting controls how parallel execution is sequenced:

```
{
  "workflow": {
    "use_worktrees": true,
    "cleanup_worktrees": true
  }
}
```

When `use_worktrees` is true, `/gsd:execute-phase` creates a *temporary* git worktree for each parallel executor agent it spawns. Each agent gets its own private filesystem view of the same git history, rooted at a scratch directory under `.planning/worktrees/` (or wherever the config points). The agents do their work inside those scratch directories, and once the phase completes, the orchestrator walks the set and aggregates their results. If `cleanup_worktrees` is also true, the scratch directories are removed after aggregation.

Without worktrees, parallel executors would share the main checkout. That's not a theoretical problem — it's a concrete, reproducible one. Two agents editing the same file race. Two agents staging different versions of a change overwrite each other. A half-applied refactor gets committed on top of an unrelated agent's fix and the git history becomes unreadable. Worktrees solve this by giving each agent a filesystem it doesn't have to share with anyone else, while still keeping all of them rooted in the same git repository so the orchestrator can collect their output.

13.2 The STATE.md Ownership Rule

This is the critical constraint. Get it right, and parallel execution is safe. Get it wrong, and you reintroduce the lost-update bug that Issue #1571 was filed to fix.

The orchestrator owns STATE.md and ROADMAP.md. Parallel executor agents must **not** write to these shared planning files directly. Each agent writes a plan-local **SUMMARY.md** inside its own worktree. After each wave, the orchestrator reads the SUMMARY.md files from every worktree and aggregates them into STATE.md and ROADMAP.md in the main checkout. This single-writer model is the entire reason parallel GSD execution is safe. Removing it — even “just this once, for a small edit” — reintroduces the lost-update bugs that corrupted shared state before the fix in Issue #1571 landed.

The rule is small enough to state in one sentence: *only the orchestrator writes to STATE.md or ROADMAP.md*. Everything else in the design follows from that constraint.

13.3 What Agents Actually Write: SUMMARY.md

Each parallel executor agent writes exactly one file in its worktree that is visible to the rest of the system: a plan-local SUMMARY.md. The SUMMARY.md format is intentionally small and additive. It captures:

- what the agent was asked to do (copied from its slice of the plan),
- what it actually did (files touched, tests run, tests passing),
- what it didn't do and why (blocking dependency, unclear requirement, environmental failure),
- any deviations from the plan that need to be reconciled at aggregation time.

The SUMMARY.md is written in the worktree, not in the main checkout. It lives next to the agent's work, and it describes what that agent did within its own sandbox. The orchestrator is responsible for reading it and deciding what to do about it. The agent never tries to speak on behalf of the plan as a whole, and the agent never writes into STATE.md or ROADMAP.md in the main checkout or in its own worktree. If you find yourself writing an executor that edits STATE.md, stop. You are reintroducing #1571.

13.4 Post-Wave Aggregation

After all parallel executors in a wave finish, the orchestrator runs an aggregation step. It:

1. Enumerates the worktrees created for this wave.
2. Reads each worktree's SUMMARY.md.

3. Reconciles the set against the master plan — which slices completed, which slices deviated, which slices blocked.
4. Updates STATE.md and ROADMAP.md in the main checkout with the aggregated result. This is the *single* write to the shared planning state for the entire wave.
5. If `cleanup_worktrees` is true, removes the worktrees and prunes the git worktree registry.

The important property is that STATE.md goes from pre-wave to post-wave in exactly one transition, authored by one process (the orchestrator), reflecting the aggregate of what all the executors did. There is never a window where two writers are touching STATE.md at the same time, because there is only ever one writer.

13.5 The `git clean` Prohibition

As of GSD v1.32 (Issue #2075), the executor wrapper **blocks** `git clean` in worktree context. Do not try to work around this. A stray `git clean -fd` can delete the prior wave's output before the orchestrator has had a chance to aggregate it — including SUMMARY.md files that were the only record of what just happened. If you need to remove untracked files inside a worktree, do it explicitly per-file (`rm path/to/file`), not en masse. The prohibition is there because the failure mode is silent: you lose work and don't notice until the aggregation step asks for a SUMMARY.md that no longer exists on disk.

The prohibition applies to any variant of the command — `git clean -f`, `git clean -fd`, `git clean -fdx` — and the wrapper will reject the call with a diagnostic that explains why. If you have a legitimate need to remove large quantities of untracked files from a worktree, the right move is to finish the wave, let the orchestrator aggregate, verify that STATE.md is up to date, and *then* clean up outside of the worktree mechanism.

13.6 Toggling Worktrees Off

Sometimes you don't want worktree-based parallel execution. Setting `workflow.use_worktrees` to false runs every executor in the main checkout, serially. Use this when:

- Your environment doesn't support git worktrees cleanly. Some CI runners and some older Windows setups have limitations that make worktrees flaky. Serial execution is more boring but more predictable in those environments.
- You're debugging and want a single deterministic file tree. Worktrees make it harder to `cd` into "the" checkout and poke around, because there are several checkouts. Turning them off is a reasonable diagnostic step.
- You intentionally want sequential execution — for example, on a very small machine where the memory cost of running three agents in parallel is worse

than the wall-clock cost of running them in series.

The trade-off is loss of parallelism. Phases will take longer to execute. The STATE.md ownership rule still applies in serial mode — the orchestrator is still the only writer — but the risk of races in the main checkout goes away because there is never more than one executor running at a time. If you find yourself fighting worktree edge cases and you don't specifically need parallelism, the boring serial path is a legitimate choice and not a defeat.

The orchestrator-only-writes-STATE.md rule is what makes parallel GSD execution safe. If you ever find yourself tempted to have an executor agent write directly to STATE.md or ROADMAP.md — “just for this one case, to update a small field” — stop and reread Issue #1571. You are about to reintroduce the bug that the current design exists to prevent.

Chapter 14

Team Patterns for Multi-Repo GSD

The previous chapters were about mechanics. This chapter is about the boring operational discipline that turns the mechanics into something a team of humans can actually live with. None of it is exotic. All of it is the kind of thing that quietly separates a team with a consistent GSD workflow from a team where every repo behaves slightly differently and no one can remember why.

14.1 Standardizing Config Across Repos

The single biggest lever for consistency is a shared `.planning/config.json` template that every repo in your team copies in. Without a template, every repo ends up with a slightly different configuration — one has worktrees enabled, another doesn't, a third uses a different `project_code` prefix, a fourth has three agent skill paths the others don't know about. The teams don't notice at first, and then six months later somebody asks “why does the billing repo's GSD output look different from the accounts repo?” and the answer is a sprawl of micro-decisions no one wrote down.

The fix is to commit a canonical template to a shared `gsd-templates` repo and pull it into every new project via a setup script. The template should pin the fields that matter for team consistency:

- `workflow.use_worktrees` — typically true for teams that execute phases in parallel.
- `workflow.cleanup_worktrees` — typically true to avoid accumulating stale worktree directories.
- `project_code` — the short prefix used for phase directories. Leave this as a template variable so each repo substitutes its own code at setup time.
- Any shared agent-skill paths — if your team has a library of skills living in a shared directory, the paths belong here so every repo picks them up.
- Any shared verifier or reviewer configuration that your team applies uniformly.

What should *not* go in the team template is anything specific to a single repo: repo-specific build commands, local credentials, one-off overrides. Those belong in the repo's own config, layered over the template.

14.2 MCP Server Scoping, Restated

Chapter 12 touched on this in passing and it's worth restating because it's the single most common mistake teams make with MCP in a multi-repo context: do not try to maintain a single global `~/.claude.json` for a microservice fleet. Use project-scoped `.mcp.json` files, one per repo, and commit the non-sensitive parts.

The reason is that microservices are each their own context. The billing service wants a connection to the billing database. The auth service wants a connection to the auth database. If you wire both of those into a single global MCP config, every Claude Code session in every repo gets every database, and the failure modes are confusing — wrong DB picked up, permissions errors from a service that should never have been able to reach that DB in the first place, credentials leaking across contexts. Project-scoped `.mcp.json` files keep each service's tool surface minimal and make it obvious from the repo alone what external systems that service talks to.

Workspaces, as we saw in Chapter 12, compose cleanly with this pattern. Each repo inside a workspace carries its own `.mcp.json`, and a workspace doesn't reach across its constituent repos to share MCP servers.

14.3 Workstream Naming Conventions

Workstream names end up in `/gsd:workstreams` list output, in `.planning/workstreams/` directory listings, in commit messages, and occasionally in PR descriptions. A little discipline here pays off indefinitely:

- **Verb-noun format.** `add-billing`, `migrate-auth`, `refactor-orm`. Every workstream is *doing* something, and the name should make that obvious at a glance.
- **No timestamps in the name.** A workstream called `2026-04-backend-work` looks tidy at creation time and looks dreadful six months later when you have fifteen of them and none of the names tell you what the work was. Use creation timestamps via the last touched column in `/gsd:workstreams` list, not in the name.
- **Don't use a bare ticket ID as the whole name.** A workstream called `JIRA-1234` is opaque. Pair the ticket ID with a short description: `JIRA-1234-billing-flow`. You get the ticket traceability and the at-a-glance readability at the same time, and the cost is a handful of extra characters.
- **Short is better than long.** Aim for workstream names that fit in a reasonable table column. 30 characters is plenty; 60 is painful.

None of these rules are enforced by the tooling. They are the kind of thing you write down in your team's `gsd-templates` repo next to the config template and agree to

honor.

14.4 The project_code Field

A small but underrated piece of team config is the `project_code` field in `.planning/config.json`. It's a short prefix — three or four letters — that GSD uses when it names phase directories and in some default commit message templates. Setting it once:

```
{  
  "project_code": "BCS"  
}
```

means that phase directories come out looking like `BCS-phase-1-auth-refactor` instead of just `phase-1-auth-refactor`. When a human later skims a flat directory listing — or reads a commit message that references a phase — the prefix tells them immediately which project the phase belongs to. In a single-repo workflow the prefix is mostly decorative. In a multi-repo team workflow, where a reviewer might be looking at phases from several different services in the same week, the prefix is a small but constant cue that prevents cross-project confusion.

Pick short, memorable codes: `BCS` for `billing-core-service`, `PAY` for `payments`, `ADM` for `admin-dashboard`. Three or four characters, upper-case, no punctuation. Record the mapping somewhere humans can find it — a short table in your team `README` or in the `gsd-templates` repo works fine.

14.5 Ticket-Based Phase Identifiers

GSD supports tagging phases with external ticket identifiers — the Jira issue, the Linear ticket, the GitHub issue number, whatever your team tracks work in. The mechanism is to record the ticket reference as phase metadata at planning time, and GSD will carry it through to commit message templates and PR descriptions automatically.

The details of which config key holds the ticket reference are version-specific and worth checking against your GSD release's documentation before you commit to a convention. What matters at the team level is the *pattern*: choose one ticket system, pick one field name, put it in the team config template, and make sure every planner is using it. When it's set up, you get traceable commits without any extra effort from the people doing the work — the ticket ID threads through from plan to commit to PR automatically.

The payoff shows up when you're looking at a six-month-old commit and wondering why it exists. If the commit says "implement session token rotation" and the message also carries `PAY-1734`, you click through to the ticket and get the full context. Without that thread, you're archaeologizing.

Standardize the boring parts — config template, naming conventions, MCP scoping, ticket integration — and your team’s GSD output will look consistent across repos even when different humans are driving different services. The interesting parts — what to plan, how to scope phases, when to escalate a workstream to a workspace — stay where they belong: in the hands of the people doing the work. Operational consistency and creative autonomy are not in tension. They live at different layers of the stack, and a well-set-up team separates them cleanly.

Appendix A

Runtime Compatibility Matrix

A consolidated lookup of every coding-agent runtime that GSD's installer targets. Use this when you need to verify the install path, the verify command, or the workspace/MCP support level for a runtime without paging back through Part I.

Runtime	Skills mat	For-	Install Dir (global)	Verify	MCP	Workspace
Claude Code 2.1.88+	skills/gsd-*/SKILL/claude/skills/		~/claude/skills/	/gsd:help	yes	yes
Claude Code (legacy)	commands/gsd-*/cmd/claude/commands/		~/claude/commands/	/gsd:help	yes	yes
Codex	skills/gsd-*/SKILL/codex/skills/		~/codex/skills/	\$gsd-help	partial	yes
Copilot	prompts + agents		~/github/	/gsd:help	partial	no
Cursor	skills/gsd-*/SKILL/cursor/		~/cursor/	/gsd:help	yes	yes
Windsurf	markdown transform		~/windsurf/	/gsd:help	partial	yes
OpenCode	config file		~/config/opencode/	/gsd-help	yes	yes
Gemini CLI	skills		~/gemini/	/gsd:help	yes	yes
Antigravity	skills		~/gemini/antigravity/	/gsd:help	yes	yes
Cline	.clinerules		~/cline/	auto-loaded	no	partial
Augment	skills form	trans-	~/augment/	/gsd:help	partial	yes
Qwen Code	skills standard)	(open	~/qwen/skills/	/gsd:help	yes	yes

Notes. The MCP column reflects whether the host runtime exposes Model Context Protocol tool surfaces to its agents. `partial` means MCP is supported via configuration but the host's discovery story is limited (e.g., Cline reads its rules file but does not expose MCP tools the same way Claude Code does). The `Workspace` column tracks GSD multi-repo workspace support; `partial` means the runtime accepts GSD's installed skill files but does not run worktree-based phases.

Antigravity has both a global install directory and a per-project `./.agent/` directory. GSD's installer auto-detects which to use based on whether you run the install script with `--global` or from inside a project directory.

Appendix B

MCP Server Quick-Reference

A scan-friendly table of MCP servers used in this guide, plus a couple of honourable mentions you'll reach for early. Each row lists the package, the transport, the auth model, and the minimal config snippet you'd drop into `.mcp.json`.

Server	Package	Transport	Auth	Minimal Config Snippet
Google Stitch	@_daveidast/stitch-mcp	stdio (proxy)	OAuth / API key	{"comma
Postgres	@modelcontextprotocol/ server-postgres	stdio	connection string	{"comma
Filesystem	@modelcontextprotocol/ server-filesystem	stdio	none (path scoped)	{"comma
GitHub	@modelcontextprotocol/ server-github	stdio	PAT	{"comma
Brave Search	@modelcontextprotocol/ server-brave-search	stdio	API key	{"comma

Every snippet that contains a credential reference here uses an environment variable placeholder. Drop the snippet into a real `.mcp.json` only after exporting the corresponding variable in your shell, and add `.mcp.json` to `.gitignore` if it carries any credentials.

Choosing a transport. `stdio` is the right default for tools that live next to your code. Use HTTP transport (`--transport http`) for cloud services and shared team servers; the GSD User Guide and the project's `docs/USER-GUIDE.md` cover the HTTP setup in more depth.

Appendix C

Workspace Command Cheat Sheet

The dense version. Print this page and tape it next to your monitor.

Workstreams (single-repo, isolated planning state)

```
/gsd:workstreams create <name>      Create a new workstream
/gsd:workstreams switch <name>      Switch the active workstream
/gsd:workstreams list                Show all workstreams (active + archived)
/gsd:workstreams complete <name>    Archive a workstream when done
```

State lives under `.planning/workstreams/<name>/`. Switching is fast - GSD just swaps which subtree it reads.

Workspaces (multi-repo or full-isolation)

```
/gsd:new-workspace --name <name> --repos <repo1,repo2,...>
                        [--strategy worktree|clone]
/gsd:list-workspaces
/gsd:remove-workspace <name>
```

Strategies:

- `worktree` (default) - git worktrees, lightweight, shares git history. Cannot have two worktrees on the same branch.
- `clone` - full git clone, complete isolation, higher disk cost. Use for spikes and throw-aways.

Workspaces live under `~/gsd-workspaces/<name>/` with a `WORKSPACE.md` manifest at the root.

```
/gsd:remove-workspace refuses removal if any constituent repo has uncommitted
changes. Commit, stash, or push first.
```

Relevant config keys

```
.planning/config.json:

workflow.use_worktrees: true | false
  Per-phase parallel executor isolation via git worktrees.

workflow.cleanup_worktrees: true | false
  Whether the orchestrator removes worktrees after phase completion.

project_code: "<prefix>"
  Short prefix used in phase directory names and commit messages.

agent_skills:
  executor:  ["/skills/<dir>", ...]
  planner:   ["/skills/<dir>", ...]
  researcher: ["/skills/<dir>", ...]
  Inject custom instruction directories into specific subagent types.
```

The single rule that makes parallel execution safe

The orchestrator owns STATE.md and ROADMAP.md. Parallel executor agents write only their plan-local SUMMARY.md. The orchestrator aggregates after each wave. Issue #1571 documents what happens when this rule is broken - don't be the one who reintroduces it.