

GSD

The Intermediate Guide

Master the full workflow. Build with confidence.

Authored by Tibsfox

For GSD v1.35.0

Released under CC BY-SA 4.0.

Authored by Tibsfox. GSD itself is MIT-licensed — see [gsd-build/get-shit-done](#).

Upstream: github.com/gsd-build/get-shit-done

Contents

1	Beyond the Basics	1
1.1	What this guide assumes	1
1.2	The gap between using GSD and understanding GSD	1
1.3	What you will <i>not</i> find here	2
2	How GSD Thinks — Context Engineering	3
2.1	The context rot problem	3
2.2	Files are the memory	4
2.3	Which file feeds which agent	4
2.4	Size limits and why they exist	5
3	The Planning Directory	6
3.1	Top-level layout	6
3.2	Inside a phase directory	7
3.3	What to commit, what not to	7
3.4	config.json in one page	7
4	The Full Lifecycle	9
4.1	When to use each step	10
4.2	Skipping steps: when it is and isn't OK	10
4.3	Shortcut commands	11
5	Discussion Phase Mastery	12
5.1	What discuss-phase actually does	12
5.2	Two modes: discuss vs. assumptions	13
5.3	Flags that change the flow	13
5.4	When to invest in discussion	14
5.5	Skipping discuss-phase entirely	14
6	Planning Deep Dive	16
6.1	Four researchers in parallel	16
6.2	The Nyquist validation layer	16
6.3	The planner and the plan-checker loop	17
6.4	XML task format, briefly	17
6.5	Other plan-phase flags worth knowing	18
7	Execution and Waves	19

7.1	Dependency graphs and wave grouping	19
7.2	What each executor gets	19
7.3	Atomic commits: one task, one commit	20
7.4	Post-execution verification	20
7.5	Why fresh contexts beat accumulated contexts	20
8	Verification and Shipping	22
8.1	verify-work and the UAT flow	22
8.2	Auto-diagnosis when things fail	22
8.3	Code review before UAT	23
8.4	Shipping: PRs and PR bodies	23
8.5	Completing milestones	24
9	Working with Existing Code	25
9.1	gsd-map-codebase: four parallel mappers	25
9.2	How mapping informs new-project	26
9.3	Lighter-weight alternatives	26
9.4	Tips for large codebases	26
10	Session Management	28
10.1	pause-work and HANDOFF.json	28
10.2	resume-work: what gets restored	28
10.3	Long-running projects: keeping state clean	28
10.4	Context warnings and proactive pausing	29
10.5	Session reports and sharing with humans	29
11	Backlog, Seeds, and Threads	31
11.1	The backlog parking lot (999.x numbering)	31
11.2	Seeds: forward-looking ideas with triggers	32
11.3	Threads: persistent context across sessions	32
11.4	Choosing the right tool	32
12	Configuration	34
12.1	Core settings	34
12.2	Workflow toggles	34
12.3	Model profiles	36
12.4	Git branching	37
12.5	Parallelization	37
12.6	Hooks and security	38
12.7	Interactive configuration	38
13	UI Design Workflow	39
13.1	ui-phase: lock the contract	39
13.2	The 6 pillars	39
13.3	shadcn initialization gate	40
13.4	Registry safety gate	40
13.5	ui-review: retroactive visual audit	40
13.6	The ui-phase safety gate	41

14 Troubleshooting	42
14.1 “Commands not found”	42
14.2 “Plans keep failing verification”	42
14.3 “Execution fails or produces stubs”	43
14.4 “Plans seem wrong or misaligned”	43
14.5 “Discuss-phase uses technical jargon I don’t understand”	43
14.6 “STATE.md is out of sync”	43
14.7 “I don’t know what went wrong” — gsd-forensics	44
14.8 gsd-health repair	44
14.9 Systematic debugging with gsd-debug	44
14.10 Context degradation in long sessions	45
14.11 Parallel execution causes build lock errors	45
14.12 Quick recovery reference	46
A Permissions Deep Dive	47
A.1 What <code>--dangerously-skip-permissions</code> does	47
A.2 The granular alternative	48
A.3 Recommended allow lists by workflow	48
A.4 Choosing between the two	50
A.5 What GSD does on its own	51
A.6 Summary	51

Chapter 1

Beyond the Basics

You have run `/gsd-new-project`. You have watched Claude spawn researchers, seen a `.planning/` directory appear, answered discussion questions, executed a plan, and shipped something. GSD worked. You liked it. You came back.

This guide is what comes next.

1.1 What this guide assumes

You are comfortable in a terminal. You know what a git commit is and why conventional commits matter. You have used Claude Code (or another GSD-supported runtime like OpenCode, Gemini CLI, Kilo, or Codex) for more than a toy project. You have completed at least one full GSD phase — ideally a full milestone — and now you want to know *how the thing actually works* so you can stop wrestling it and start steering it.

If any of that sounds too advanced, the Beginner’s Guide will serve you better. Come back here after your first shipped milestone.

1.2 The gap between using GSD and understanding GSD

The first time you run `/gsd-plan-phase`, four researchers fire off in parallel, a planner runs, a plan-checker reviews the plan up to three times, and a `PLAN.md` file appears in your phase directory. If you stopped and asked yourself at each step “what is this agent reading? what is it writing? why does it spawn *here* and not later?” you would have a working mental model of GSD.

Most people do not stop. Most people just run the command and look at the output. That works until the day it doesn’t — until a plan comes back wrong, or a wave of parallel executors collides on a file, or `STATE.md` drifts from reality and suddenly `/gsd-progress` is lying to you. When that day arrives you have two options: dig in and learn, or blame the tool.

This guide is the dig-in option. By the end of it, you will understand:

- Why GSD maintains the file ecosystem it does, and which file feeds which agent.
- What each of the six main workflow stages actually does — not just its name.
- When to trust the defaults and when to override them.
- How to recover when something goes sideways, without nuking `.planning/` and starting over.
- How to configure GSD for your team, your budget, and your tolerance for friction.

1.3 What you will *not* find here

This is not the architecture deep dive. If you want the orchestrator-subagent internals, the XML prompt structure, the security model at the code level, team adoption playbooks, or SDK automation, that lives in the Advanced Guide. If you want a quick flag lookup, the Command Reference is a better tool than flipping through chapters.

This guide sits in the middle on purpose. It explains *enough* mechanism to let you troubleshoot and configure with confidence, without drowning you in implementation detail you will never touch.

- Your target: understand *why* GSD creates the files it does, not just that it does.
- Every chapter that follows works standalone — skip around.
- When something goes wrong, the failure mode almost always traces back to one of the files we are about to explain.

Chapter 2

How GSD Thinks --- Context Engineering

GSD calls itself a meta-prompting framework. What that actually means is: GSD is an opinionated pipeline for turning a vague idea into a series of focused prompts, each handed to a fresh AI agent with exactly the context it needs and nothing more. Understanding that sentence is understanding GSD.

2.1 The context rot problem

Every AI coding agent has a context window — roughly 200,000 tokens for current frontier models, up to 1M for Opus 4.6 and Sonnet 4.6 under the right conditions. As you work, that window fills with conversation history, tool outputs, file reads, and system prompts. Past a certain point, quality degrades: the model starts forgetting earlier decisions, contradicting itself, hallucinating file paths, and confusing one feature for another. The community calls this *context rot*.

Long chat sessions rot. Long-running agent tasks rot. The cure that every power user eventually discovers is “just start a new session.” GSD bakes that insight into the workflow at the structural level.

WITHOUT GSD (accumulated context)	WITH GSD (fresh context per agent)		
[200K window]	[200K]	[200K]	[200K]
conversation 1	researcher	planner	executor
+ conversation 2			
+ conversation 3	v	v	v
+ tool output	RESEARCH	PLAN.md	SUMMARY
+ tool output	.md		
+ ...			
+ QUALITY DROPS			

The first time you see GSD’s wave execution, this is the design choice you’re watching: the orchestrator spawns one agent per plan with a fresh 200K window, and when the agent finishes its atomic task it terminates. No accumulated history. No

rot. The next wave spawns fresh agents with fresh windows, reading only the compact artifacts they need.

2.2 Files are the memory

If agents are ephemeral, where does project memory live? On disk, as Markdown. GSD's core bet is that human-readable files are the right state store for an AI workflow — inspectable by you, inspectable by every agent, survivable across context resets (/clear), and version-controllable with git.

The file ecosystem has a deliberate shape:

```
.planning/
|
+-- PROJECT.md          <-- vision, constraints, evolution rules
+-- REQUIREMENTS.md    <-- v1 / v2 / out-of-scope scope IDs
+-- ROADMAP.md         <-- phases and their status
+-- STATE.md           <-- "where am I right now?" living memory
+-- config.json        <-- workflow toggles, model profile, gates
+-- MILESTONES.md     <-- completed milestone archive
+-- research/         <-- from /gsd-new-project
+-- codebase/         <-- from /gsd-map-codebase (brownfield)
+-- phases/
    +-- 01-auth-setup/
        +-- 01-CONTEXT.md      <-- discuss-phase output
        +-- 01-RESEARCH.md     <-- plan-phase researcher output
        +-- 01-VALIDATION.md   <-- Nyquist test-coverage contract
        +-- 01-UI-SPEC.md      <-- ui-phase design contract
        +-- 01-01-PLAN.md      <-- planner output
        +-- 01-02-PLAN.md      <-- executor output
        +-- 01-01-SUMMARY.md   <-- executor output
        +-- 01-VERIFICATION.md <-- verifier output
        +-- 01-UAT.md          <-- verify-work output
        +-- 01-REVIEW.md       <-- code-review output
```

Each file is both an input and an output. CONTEXT.md is *written* by discuss-phase and *read* by the researcher, planner, and executor. PLAN.md is written by the planner and read by the executor and plan-checker. Nothing is coincidental — every artifact has a producer and a set of consumers.

2.3 Which file feeds which agent

This is the mapping worth memorizing. When a plan goes wrong, trace backwards: which file shaped the decision? When a researcher returns weak findings, ask: did CONTEXT.md give it enough to work with?

File	Consumed by (roughly)
PROJECT.md	Every agent. The always-loaded north star.
REQUIREMENTS.md	Planner, verifier, auditor.
ROADMAP.md	Every orchestrator that needs to know the phase sequence.
STATE.md	Every agent. Decisions, blockers, position, session memory.
CONTEXT.md (per phase)	Researcher, planner, executor.
RESEARCH.md (per phase)	Planner, plan-checker.
VALIDATION.md (per phase)	Planner, executor (test coverage contract).
UI-SPEC.md (per phase)	Executor, UI auditor.
PLAN.md (per plan)	Executor, plan-checker.
SUMMARY.md (per plan)	Verifier, state tracking, next-wave executors.

2.4 Size limits and why they exist

PROJECT.md and STATE.md are loaded into every agent's context. If they balloon, every agent pays the tax — and you lose the context budget you wanted reserved for the actual work. GSD's templates intentionally keep these files concise. A PROJECT.md that reaches 300 lines is usually hiding decisions that belong in a CONTEXT.md for a specific phase; a STATE.md over 200 lines usually has old session notes that should have been archived into MILESTONES.md.

If a file gets big and your plans start looking vague, that is the failure mode to suspect.

These files *are* the context. Edit them carefully. A sloppy edit to PROJECT.md propagates to every future agent spawn on the project. If you are going to hand-edit, prefer CONTEXT.md for a specific phase — its blast radius is scoped.

- Fresh contexts beat accumulated contexts. GSD's wave model is how it operationalizes that.
- File-based state is what makes context resets safe.
- Every artifact has a known producer and known consumers — troubleshooting starts by naming the file.
- Keep PROJECT.md and STATE.md lean. They tax every agent.

Chapter 3

The Planning Directory

This chapter is a guided tour of *.planning/*. Beginners treat it as a black box. Intermediate users open it up.

3.1 Top-level layout

.planning/ is created by `/gsd-new-project` and grows as you work. The top-level files are stable — you will see them on every project — and the subdirectories appear on demand.

File / directory	Role
<i>PROJECT.md</i>	Project vision, constraints, decisions that outlive phases.
<i>REQUIREMENTS.md</i>	Scoped requirements with IDs (v1 / v2 / out-of-scope).
<i>ROADMAP.md</i>	Phase breakdown with per-phase Depends on and status.
<i>STATE.md</i>	Living memory: current position, decisions, blockers, metrics.
<i>config.json</i>	Workflow toggles, model profile, gates, git strategy.
<i>MILESTONES.md</i>	Archive of completed milestones.
<i>HANDOFF.json</i>	Written by <code>/gsd-pause-work</code> , restored by <code>/gsd-resume-work</code> .
<i>research/</i>	Project-level research from <code>/gsd-new-project</code> .
<i>codebase/</i>	Brownfield maps from <code>/gsd-map-codebase</code> or <code>/gsd-scan</code> .
<i>phases/</i>	Per-phase directories — the bulk of active work.
<i>seeds/</i>	Forward-looking ideas with trigger conditions (<code>/gsd-plant-seed</code>).
<i>threads/</i>	Lightweight persistent context threads (<code>/gsd-thread</code>).
<i>todos/</i>	Captured ideas (pending/ and done/).
<i>debug/</i>	Active and resolved debug sessions.
<i>reports/</i>	Session reports from <code>/gsd-session-report</code> .
<i>forensics/</i>	Diagnostic reports from <code>/gsd-forensics</code> .
<i>intel/</i>	Optional queryable codebase intelligence index.
<i>ui-reviews/</i>	Screenshots from <code>/gsd-ui-review</code> (auto-gitignored).
<i>learnings/</i>	Extracted patterns from <code>/gsd-extract-learnings</code> .

3.2 Inside a phase directory

Phase directories are where most of the action happens. They follow a naming convention — zero-padded phase number, a hyphen, a slug. For an `auth-setup` phase at position 1 you get `01-auth-setup/`. If you configured `project_code` in `config.json`, phase dirs get a prefix like `ABC-01-auth-setup/`.

A typical phase directory mid-work:

```

phases/01-auth-setup/
|
+-- 01-CONTEXT.md          <-- /gsd-discuss-phase 1
+-- 01-UI-SPEC.md          <-- /gsd-ui-phase 1      (frontend phases)
+-- 01-RESEARCH.md        <-- /gsd-plan-phase 1 (researchers)
+-- 01-VALIDATION.md      <-- /gsd-plan-phase 1 (Nyquist auditor)
+-- 01-01-PLAN.md         <-- /gsd-plan-phase 1 (planner)
+-- 01-02-PLAN.md         <-- /gsd-plan-phase 1 (planner)
+-- 01-01-SUMMARY.md      <-- /gsd-execute-phase 1 (executor A)
+-- 01-02-SUMMARY.md      <-- /gsd-execute-phase 1 (executor B)
+-- 01-VERIFICATION.md    <-- /gsd-execute-phase 1 (verifier)
+-- 01-UAT.md             <-- /gsd-verify-work 1
+-- 01-REVIEW.md          <-- /gsd-code-review 1
+-- 01-UI-REVIEW.md       <-- /gsd-ui-review 1
+-- 01-SECURITY.md        <-- /gsd-secure-phase 1 (optional)
+-- 01-DISCUSSION-LOG.md  <-- audit trail of discuss-phase

```

Two number conventions live here. The outer two-digit number is the *phase* number (01). Plans inside a phase get a second two-digit number (01-01-PLAN.md, 01-02-PLAN.md). Plans are the unit of parallel execution — each plan in a wave gets its own executor.

3.3 What to commit, what not to

By default, `.planning/` commits to git. GSD treats planning artifacts as part of the project's history — phase intent is as valuable to the team as the code it produced. Two knobs change this:

- `planning.commit_docs` — set to `false` to stop committing. The installer auto-detects this: if `.planning/` is already in `.gitignore`, it treats `commit_docs` as `false` regardless of the config value.
- `planning.search_gitignored` — set to `true` so broad searches (for example, `grep` through the repo) still find planning files even when they are gitignored.

Keep planning commits on for team projects where shared visibility matters. Turn them off for sensitive/private work or if your team actively dislikes planning noise in diffs. Either way, `/gsd-pr-branch` gives you a planning-free branch for PR review.

3.4 config.json in one page

`.planning/config.json` is the single source of truth for how GSD behaves on this project. It lives inside `.planning/`, so per-project settings are self-documenting;

global defaults live at `~/.gsd/defaults.json` and are merged in at project creation time with per-project values winning.

The skeleton you will see most often:

```
{
  "mode": "interactive",
  "granularity": "standard",
  "model_profile": "balanced",
  "planning": {
    "commit_docs": true,
    "search_gitignored": false
  },
  "workflow": {
    "research": true,
    "plan_check": true,
    "verifier": true,
    "nyquist_validation": true,
    "ui_phase": true,
    "ui_safety_gate": true,
    "discuss_mode": "discuss",
    "skip_discuss": false,
    "code_review": true
  },
  "git": {
    "branching_strategy": "none"
  },
  "parallelization": {
    "enabled": true,
    "max_concurrent_agents": 3
  },
  "hooks": {
    "context_warnings": true,
    "workflow_guard": false
  }
}
```

Every key is optional. Missing keys default to true for workflow toggles (the “absent equals enabled” rule) and to sensible defaults for everything else. Chapter 12 covers each group in depth.

- `.planning/` is not a black box. Open it and read the files.
- Phase directories follow a strict naming convention. The numbers matter.
- `planning.commit_docs` controls whether planning joins git history. Pick intentionally.
- `config.json` is the project’s control panel.

Chapter 4

The Full Lifecycle

The headline GSD workflow is a loop. One pass through the loop handles one phase:

```
/gsd-discuss-phase <-- capture your preferences
|
/gsd-ui-phase      <-- (optional, frontend only) design contract
|
/gsd-plan-phase   <-- research + plan + verify loop
|
/gsd-execute-phase <-- parallel wave execution
|
/gsd-code-review  <-- (optional) structured review
|
/gsd-verify-work  <-- UAT with auto-diagnosis
|
/gsd-ship         <-- create PR (optional)
|
next phase? yes --> (loop back to discuss-phase)
| no
v
/gsd-audit-milestone
/gsd-complete-milestone
|
/gsd-new-milestone <-- start the next version
```

This loop is the spine of every chapter that follows. Each stage is a separate command that reads specific files, writes specific files, and can be run or re-run in isolation. That separation is not bureaucracy — it is what lets you recover surgically when something breaks.

4.1 When to use each step

Stage	Use when
<code>/gsd-discuss-phase</code>	You haven't locked in implementation decisions for the phase. This is the single biggest lever on plan quality. Skipping it guarantees Claude fills the gaps with its own guesses.
<code>/gsd-ui-phase</code>	The phase builds frontend UI. The design contract prevents "five components, five visual languages" drift.
<code>/gsd-plan-phase</code>	After discussion is locked. Runs research (4 parallel agents) then planner then plan-checker loop.
<code>/gsd-execute-phase</code>	After planning is verified. Runs parallel executor waves.
<code>/gsd-code-review</code>	After execution, before verification, when you want a structured bug/security pass. Configurable via <code>workflow.code_review_depth</code> .
<code>/gsd-verify-work</code>	After execution (and optional code review). Manual UAT with auto-diagnosis — the human quality gate.
<code>/gsd-ship</code>	After UAT passes. Creates a GitHub PR with an auto-generated body.
<code>/gsd-audit-milestone</code>	When all phases in a milestone are done. Catches gaps before you tag a release.
<code>/gsd-complete-milestone</code>	When the audit is clean. Archives and tags.
<code>/gsd-new-milestone</code>	When you are ready to start the next version cycle. Scans seeds, updates <code>PROJECT.md</code> , generates new <code>REQUIREMENTS.md</code> / <code>ROADMAP.md</code> .

4.2 Skipping steps: when it is and isn't OK

Not every stage runs on every phase. Some combinations are fine, others get you into trouble.

Safe to skip:

- `/gsd-ui-phase` on backend-only phases. GSD detects this and won't nag.
- `/gsd-code-review` when the phase touched trivial code and verification caught the issues you care about. Or when you plan to rely on `/gsd-audit-fix` later.
- `/gsd-ship` when you don't use GitHub PRs for this repo.

Skip at your own risk:

- `/gsd-discuss-phase` — unless the phase is genuinely well-scoped from `ROADMAP.md` alone, this is where plan quality is made or broken. If you must skip, set `workflow.skip_discuss: true` and rely on a comprehensive `PROJECT.md`.

- `/gsd-plan-phase` — there is no safe skip. You can disable research (`--skip-research`) or plan-checker (`--skip-verify`) inside it, but the planner itself is not optional.
- `/gsd-verify-work` — skipping the UAT is skipping the human-in-the-loop. On anything user-facing, do not.

Do not skip:

- `/gsd-execute-phase`. That is where code gets written.
- `/gsd-audit-milestone` before `/gsd-complete-milestone` when shipping to production.

4.3 Shortcut commands

Two commands exist to bypass the full loop for trivial work:

- `/gsd-quick` — ad-hoc task with GSD guarantees. Flags `--discuss`, `--research`, and `--full` let you re-opt-in to discussion, research, and the full plan-check-verify pipeline. Good for bug fixes, small features, config changes.
- `/gsd-fast` — inline trivial tasks. No subagents, no planning, nothing. Good for typos, config tweaks, forgotten commits. Not a replacement for `/gsd-quick`.

And one command for when you do not want to think about which stage is next:

- `/gsd-next` — auto-detects your current state and runs the appropriate command. Great for “I came back from lunch, where was I?”

- The workflow is a loop of six core commands per phase, plus milestone wrap-up commands.
- Each stage is independently re-runnable — that separation is what makes recovery possible.
- Discussion and UAT are where most users are tempted to skip. Don't, unless you know exactly why.
- `/gsd-next` is your friend when you are not sure what to run.

Chapter 5

Discussion Phase Mastery

`/gsd-discuss-phase` is the stage most people underestimate. It looks like a formality — “just answer some questions so Claude knows what I want.” It is actually the single highest-leverage stage in the whole workflow. Every file downstream is shaped by `CONTEXT.md`. A thoughtful discussion produces a plan that you don’t have to fight. A rushed discussion produces a plan you spend `/gsd-verify-work` rejecting.

5.1 What discuss-phase actually does

Under the hood, `discuss-phase` spawns an advisor agent that reads `PROJECT.md`, `ROADMAP.md`, the current phase’s goal, and any existing `CONTEXT.md` (if you are re-running). It surfaces the phase’s “gray areas” — decisions the planner will need to make, which you haven’t made yet — and asks you to resolve them.

The output is `{phase}-CONTEXT.md`, a structured file with six sections:

Section	Holds
<code><domain></code>	The phase boundary — what’s in scope and what isn’t.
<code><decisions></code>	Locked implementation decisions (libraries, patterns, trade-offs).
<code><canonical_refs></code>	Specs, docs, or upstream references agents must read.
<code><code_context></code>	Reusable assets, patterns, integration points from existing code.
<code><specifics></code>	Your preferences and specific guidance.
<code><deferred></code>	Ideas noted for future phases.

A full audit trail lands in `{phase}-DISCUSSION-LOG.md`, so you can see what was asked and what you answered.

5.2 Two modes: discuss vs. assumptions

GSD has two ways to run discuss-phase. The default is `discuss` — open-ended interview-style questions, grouped by gray area. The alternative is `assumptions`, introduced in v1.28. You switch via `/gsd-settings` or directly in `config.json`:

```
"workflow": {  
  "discuss_mode": "assumptions"  
}
```

Discuss mode (the default):

- Claude identifies gray areas, presents them for selection, then asks ~4 questions per area.
- Best for early phases, new codebases, or when you have strong opinions to express proactively.
- Typical interaction count: 15-20.

Assumptions mode:

- Claude deeply analyzes the codebase via a subagent (reads 5-15 relevant files).
- Forms assumptions with evidence and a confidence level (Confident / Likely / Unclear).
- Presents them for confirmation or correction — you only touch the ones that are wrong.
- Best for established codebases with clear patterns.
- Typical interaction count: 2-4.

The output `CONTEXT.md` is identical in both modes. Downstream agents do not care which path you took.

5.3 Flags that change the flow

`/gsd-discuss-phase` accepts several flags that reshape how questions are asked:

Flag	Effect
<code>--auto</code>	Auto-select the recommended default for every question. In assumptions mode, skips the confirm gate and auto-resolves anything marked Unclear.
<code>--batch</code>	Group questions into batches instead of one-by-one. Faster for power users.
<code>--analyze</code>	Add a trade-off analysis panel for each question. Slower, better-reasoned.
<code>--power</code>	File-based bulk answering from a prepared answers file. Use when you have a PRD or spec you can translate into answers programmatically.
<code>--text</code>	Plain-text numbered lists instead of TUI menus. Required for remote Claude Code sessions (<code>/rc</code> mode).

And the top-level `--chain` on `/gsd-quick` (not `discuss-phase`) auto-advances through `discuss` → `plan` → `execute`. Composable with `--discuss` and `--research` on `/gsd-quick`.

5.4 When to invest in discussion

Not every phase deserves 20 minutes of interview. A rough calibration:

- **First phase of a new milestone.** Invest heavily. This is where architecture decisions compound. Use standard `discuss` mode, no `--auto`, with `--analyze` if the phase touches anything you are unsure about.
- **Mid-milestone phases in familiar territory.** Assumptions mode shines. Most of your conventions are already locked in from earlier phases.
- **Brownfield phases.** Run `/gsd-map-codebase` first, then use assumptions mode — Claude now has a full map to form assumptions from.
- **Bug-fix phases.** `/gsd-quick --discuss` is usually enough. Or `/gsd-discuss-phase --auto` if defaults are probably right.
- **Speculative / R&D phases.** Use `/gsd-explore` first for Socratic ideation, then `/gsd-discuss-phase --analyze` to pin down tradeoffs.

5.5 Skipping discuss-phase entirely

There is a switch to opt out of discussion altogether:

```
"workflow": {
  "skip_discuss": true
}
```

When `skip_discuss` is true, running `/gsd-autonomous` bypasses `discuss-phase` entirely. A minimal `CONTEXT.md` is generated from the `ROADMAP` phase goal alone. This is intended for projects where `PROJECT.md` and `REQUIREMENTS.md` are

comprehensive enough that discussion adds no new information — usually autonomous multi-phase runs on well-understood work.

If your plans come back vague or misaligned, `/gsd-discuss-phase` is almost always the fix. Most “GSD planned the wrong thing” complaints trace back to a shallow or skipped discussion.

- `CONTEXT.md` shapes everything downstream. Treat discuss-phase as a primary product, not a ceremony.
- Assumptions mode trades an interview for a confirmation pass. Use it on established codebases.
- `--auto`, `--batch`, `--analyze`, `--power` reshape pacing without changing the output format.
- Skip discussion only with `workflow.skip_discuss: true` and a *very* complete `PROJECT.md`.

Chapter 6

Planning Deep Dive

`/gsd-plan-phase` is the second-highest leverage stage after discussion. It runs three things back to back: research, planning, and verification. Each has moving parts worth understanding.

6.1 Four researchers in parallel

When research is enabled (`workflow.research: true`, which is the default), `/gsd-plan-phase` spawns four researcher agents in parallel, each with a fresh context window and a narrow mandate:

- **Stack researcher** — investigates the tech stack, libraries, language features, and constraints specific to the phase.
- **Features researcher** — investigates what this kind of feature usually entails, what's table stakes, what's differentiating.
- **Architecture researcher** — investigates patterns, data flow, component boundaries, and integration points.
- **Pitfalls researcher** — investigates known failure modes, edge cases, security considerations, and things people routinely get wrong.

Outputs land in `{phase}-RESEARCH.md`. This file is then consumed by the planner and the plan-checker.

If you want to skip this step (you know the domain cold, or the phase is mechanical), use `--skip-research`. If you want to force re-research on an existing phase, use `--research`.

6.2 The Nyquist validation layer

Since v1.27, GSD runs an 8th plan-check dimension that looks specifically at test coverage. The Nyquist auditor maps each requirement in the phase to a specific


```

<description>
  Add JWT signing utility in src/auth/jwt.ts.
</description>
<acceptance>
  - Exports signAccessToken() and verifyAccessToken()
  - Uses HS256 for dev, RS256 for prod (from env)
</acceptance>
<files>
  - src/auth/jwt.ts (new)
  - src/auth/jwt.test.ts (new)
</files>
<verify>npm test -- src/auth/jwt.test.ts</verify>
</task>

```

The executor reads these and implements each task atomically — one task, one commit. The XML is not for you to read — it’s for the executor agent to parse reliably. But when a plan comes back wrong it is often instructive to open *PLAN.md* and scan the task XML. You can see exactly what the planner thought you wanted.

6.5 Other plan-phase flags worth knowing

Flag	Effect
--gaps	Gap-closure mode. Reads an existing VERIFICATION.md and targets remaining gaps without re-running research.
--prd <file>	Use a PRD file instead of CONTEXT.md as the source of truth.
--reviews	Re-plan with cross-AI review feedback from REVIEWS.md (produced by <i>/gsd-review</i>).
--validate	Run state validation before planning begins. Catches STATE.md drift early.
--bounce	Run an external plan-bounce validation script after planning. Configured by workflow.plan_bounce_script.
--auto	Non-interactive: skip confirmation gates between sub-steps.

- Plan-phase is research → plan → verify, not a single monolithic step.
- Four parallel researchers cover stack, features, architecture, and pitfalls.
- The plan-checker loop catches most bad plans before they reach the executor.
- Each plan contains 2-3 atomic tasks. Larger “plans” are a smell.

Chapter 7

Execution and Waves

With plans verified, `/gsd-execute-phase N` takes over. This is where GSD's context-engineering bet pays its biggest dividend: parallel executors with fresh context windows, one per plan.

7.1 Dependency graphs and wave grouping

When you run `/gsd-execute-phase`, the orchestrator first analyzes the plans in the phase. Each plan has a `Depends on` field that says which other plans (if any) must complete before it starts. The orchestrator groups plans into *waves*: a wave is a set of plans with no unsatisfied dependencies, so every plan in a wave can run in parallel.

```
Wave Analysis example:
Plan 01 (no deps)      +-
Plan 02 (no deps)      -+-- Wave 1 (parallel: 01 and 02)
Plan 03 (depends: 01)  --- Wave 2 (waits for Wave 1)
Plan 04 (depends: 02)  +-
Plan 05 (depends: 03,04) -- Wave 3 (waits for Wave 2)
```

Within a wave, plans run concurrently (up to `parallelization.max_concurrent_agents`, default 3). Across waves, execution is sequential — wave 2 cannot start until wave 1 is fully committed.

You can execute a single wave with `--wave N`. Useful for: replaying a failed wave, manually splitting a long execution, or resuming after interruption.

7.2 What each executor gets

Every executor in a wave spawns with a fresh 200K context window — or up to 1M for 1M-class models — and receives exactly three categories of input:

1. Its assigned `PLAN.md` file. This is the authoritative task list.
2. Project context: `PROJECT.md`, `STATE.md`.

3. Phase context: CONTEXT.md, RESEARCH.md, VALIDATION.md, and UI-SPEC.md if present.

For 1M-context models (Opus 4.6, Sonnet 4.6), GSD also injects prior-wave SUMMARY.md files so later waves can see what earlier waves shipped. This is *adaptive context enrichment*. You get it automatically when you are on a 1M-class model; for 200K models, the orchestrator uses compact truncated prompts with cache-friendly ordering.

7.3 Atomic commits: one task, one commit

Every task in a plan produces one git commit. The commit message follows the project's convention (Conventional Commits, in most cases) and references the task ID. If a task fails — tests don't pass, compile errors, verification fails — the task is retried up to `workflow.node_repair_budget` times (default 2) via the node-repair agent before it escalates.

This atomicity is what makes `/gsd-undo` safe. You can revert a single task without touching neighboring work.

When parallelization is enabled, executors commit with `--no-verify` to avoid pre-commit hook contention (especially painful with Rust and cargo lock). The orchestrator runs pre-commit hooks once per wave, after all parallel commits complete. If you need hooks on every commit, set `parallelization.enabled: false`.

7.4 Post-execution verification

After all waves complete, a verifier agent runs. It reads every PLAN.md, SUMMARY.md, CONTEXT.md, and the REQUIREMENTS.md for the phase, and checks the codebase against the phase goal. It looks for:

- Code that matches the plan but doesn't match the intent.
- Tests that are disabled, circular, or weak on assertions.
- Tasks that got committed but not wired up.
- Cross-cutting concerns that slipped through individual plans.

The output is `{phase}-VERIFICATION.md`. If the verifier passes, execution is done and the phase is ready for UAT. If it fails, issues are logged for `/gsd-verify-work` to surface.

7.5 Why fresh contexts beat accumulated contexts

It's worth making the design argument explicit, because when you see the wave execution log — five different agents each starting from a blank slate, each re-reading the phase files, each writing one commit — it *looks* wasteful. You could

imagine a single agent plowing through all five tasks in one context, saving those re-reads.

Don't. Here's why:

- **Quality floor.** A fresh agent makes no decisions shaped by the noise of prior decisions. Each task gets the model's full attention, not the leftover attention after four other tasks filled the window.
- **Independence.** Parallel waves are only possible because each executor is isolated. One agent one task makes parallelism a syntactic property of the phase.
- **Recoverability.** A single accumulated agent that fails halfway through leaves a corrupted state. Five atomic agents either all commit or only some commit — you can `/gsd-undo` the partial ones and re-execute.
- **Debuggability.** When a task goes wrong, you can read the plan, look at the commit, compare them, and know exactly what that one agent did. No interleaved history to unpack.

The re-reads are the cost. They're usually cached (GSD uses cache-friendly prompt ordering to maximize prompt-cache hits). The quality win more than pays for them.

- Wave grouping is derived from plan dependencies. Plans with no deps run in wave 1.
- Each executor gets a fresh context window and a single plan. Nothing more.
- One task, one commit — that atomicity is what makes every downstream recovery command work.
- The verifier is a second pass that checks the whole phase, not just individual plans.

Chapter 8

Verification and Shipping

Execution wrote code. Verification proves the code matches intent. Shipping tells the world.

8.1 verify-work and the UAT flow

`/gsd-verify-work N` runs manual user acceptance testing for a completed phase. “Manual” is a slight misnomer — most of the work is automated, but you are in the loop for the final judgment calls.

Under the hood, `verify-work` reads `PLAN.md`, `SUMMARY.md`, `VERIFICATION.md`, and the phase’s acceptance criteria, and walks you through a structured test plan. For each criterion it asks: does the delivered code satisfy this? It will run commands, show you output, and ask for your confirmation.

The output is `{phase}-UAT.md` — a structured record of what was tested, what passed, what failed, and how each failure was classified.

8.2 Auto-diagnosis when things fail

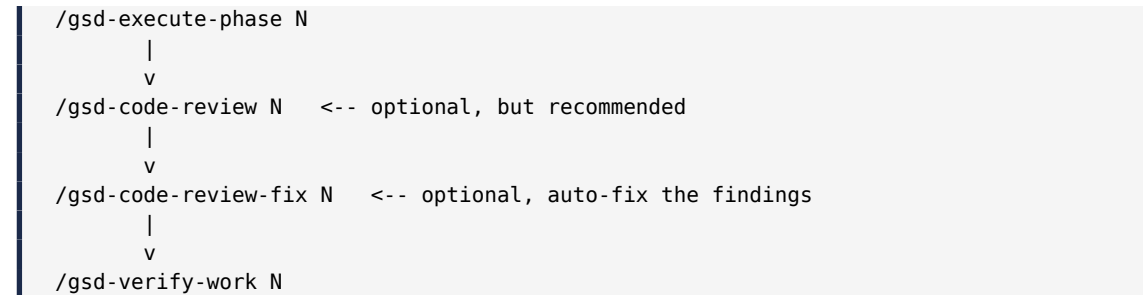
When `verify-work` finds a failure, it doesn’t just log it. An auto-diagnosis step kicks in: it reads the relevant plan, the summary, the commit history, and tries to identify the root cause. Three things can happen:

1. The diagnosis is confident enough to generate a *fix plan*. A new plan file is created under the phase and you can `/gsd-execute-phase --wave N` to run just the fix.
2. The diagnosis points at a config or context issue. You are told to re-run `/gsd-discuss-phase` or edit `CONTEXT.md` and re-plan.
3. The diagnosis is inconclusive. Escalated to you for manual decision.

Crucially, `/gsd-verify-work` never silently edits code. It proposes fixes and you run them. That’s the “user in the loop” part.

8.3 Code review before UAT

Since v1.34, GSD has a dedicated code review stage between execution and UAT:



`/gsd-code-review N` scopes files by reading `SUMMARY.md` (or git diff as fallback) and runs a reviewer agent configured for depth:

- `--depth=quick` — pattern-matching only, ~2 minutes.
- `--depth=standard` — per-file analysis with language-specific checks, ~5-15 minutes. Default.
- `--depth=deep` — cross-file analysis with import graphs and call chains, ~15-30 minutes.

Findings go into `{phase}-REVIEW.md`, classified as Critical / Warning / Info. You then run `/gsd-code-review-fix N` to auto-fix. Add `--auto` to run a fix-and-review loop (up to 3 iterations) until clean. Add `--all` to include Info-level findings in the fix scope.

Set the default depth globally via `workflow.code_review_depth` in `config.json`.

8.4 Shipping: PRs and PR bodies

`/gsd-ship N` creates a GitHub PR from the phase's commits. Prerequisites: the phase has been verified (`/gsd-verify-work` passed), and gh CLI is installed and authenticated.

The PR body is auto-generated from planning artifacts:

- The phase goal from ROADMAP.md.
- A changes summary built from all SUMMARY.md files in the phase.
- The requirements addressed (REQ-IDs from REQUIREMENTS.md).
- Verification status from VERIFICATION.md.
- Key decisions from CONTEXT.md.

Flags:

- `--draft` — create as a draft PR.

- `/gsd-ship` with a milestone version (e.g., `v1.0`) ships a whole milestone instead of a phase.

If your team reviews code without the planning noise, run `/gsd-pr-branch` first. It creates a branch that filters out `.planning/` commits so the reviewer diff is clean.

8.5 Completing milestones

Once every phase in the milestone is verified, the wrap-up sequence is:

1. `/gsd-audit-milestone` — runs one last gap check. It reads `REQUIREMENTS.md` and compares against every `SUMMARY.md` and `VERIFICATION.md` to confirm nothing slipped. Produces a gap analysis report.
2. `/gsd-plan-milestone-gaps` if gaps exist — creates one phase per gap so you can close them without re-doing audit manually.
3. `/gsd-complete-milestone` — archives the milestone into `MILESTONES.md` and tags the release in git.
4. `/gsd-new-milestone` when you are ready for the next version. It reads `PROJECT.md`, scans seeds, prompts for the new milestone's scope, and regenerates `REQUIREMENTS.md` and `ROADMAP.md`. Optionally takes `--reset-phase-numbers` to restart numbering at Phase 1.
5. `/gsd-milestone-summary` — generates a comprehensive milestone summary report for team onboarding or stakeholder review.

- Verify-work is automated UAT with you in the loop for final judgments.
- Failed UAT produces fix plans, not silent code edits.
- `/gsd-code-review` before UAT catches bugs UAT won't catch. Use `--depth=deep` on production phases.
- Auto-generated PR bodies pull directly from planning artifacts — write better plans, get better PRs.

Chapter 9

Working with Existing Code

Greenfield projects are easy. GSD shines on those. But most real work is brownfield — you are adding features to a codebase that already has conventions, architecture, technical debt, and patterns you don't want the planner to reinvent. GSD has a dedicated preflight for this case.

9.1 gsd-map-codebase: four parallel mappers

Run this *before* `/gsd-new-project` on any existing codebase:

```
| /gsd-map-codebase
```

Four mapper agents fire in parallel and write into `.planning/codebase/`:

- **Stack Mapper** → `codebase/STACK.md` — languages, frameworks, build tools, key dependencies.
- **Architecture Mapper** → `codebase/ARCHITECTURE.md` — layers, module boundaries, data flow, integration points.
- **Convention Mapper** → `codebase/CONVENTIONS.md` — naming patterns, file organization, error handling, testing style.
- **Concern Mapper** → `codebase/CONCERNS.md` — known rough spots, TODOs, deprecated APIs, “here be dragons” files.

A `/gsd-map-codebase` run on a small project takes a few minutes. On a large one, it can take longer — the four agents work in parallel but each one may read dozens of files. You can narrow the scope with an argument: `/gsd-map-codebase auth` focuses mapping on the auth area.

Two supplementary files that sometimes get produced depending on project shape:

- `codebase/STRUCTURE.md` — directory topology.
- `codebase/TESTING.md` — test framework, test locations, coverage.

- `codebase/INTEGRATIONS.md` — external services, APIs, databases.

9.2 How mapping informs new-project

With `codebase/` populated, `/gsd-new-project` behaves differently: questions focus on what you are *adding*, not on rediscovering what already exists. The project-researcher still runs, but it has the mapping to anchor against. `REQUIREMENTS.md` and `ROADMAP.md` are framed as delta work against the existing code.

```

/gsd-map-codebase
|
+-- Stack Mapper      --> codebase/STACK.md
+-- Arch Mapper       --> codebase/ARCHITECTURE.md
+-- Convention Mapper --> codebase/CONVENTIONS.md
+-- Concern Mapper    --> codebase/CONCERNS.md
|
v
/gsd-new-project
|
Questions focus on what you're ADDING
Planning automatically loads your patterns

```

9.3 Lighter-weight alternatives

`/gsd-map-codebase` is heavy — four parallel agents, lots of reads. For smaller needs, GSD has two lighter tools:

- `/gsd-scan` — single-focus codebase assessment. One mapper agent instead of four. Flags: `--focus tech`, `--focus arch`, `--focus quality`, `--focus concerns`, `--focus tech+arch` (default). Use this when you just want a quick read on one dimension.
- `/gsd-intel` — queryable codebase intelligence. Requires `intel.enabled: true` in `config.json`. Subcommands: `refresh` builds the index, `query <term>` searches across files, `status` shows freshness, `diff` shows changes since the last snapshot. Produces JSON files in `.planning/intel/` (`stack`, `api-map`, `dependency-graph`, `file-roles`, `arch-decisions`).

`intel` is closer to an index than a map — faster to query, but requires a one-time setup and periodic refreshes. Use it on codebases large enough that re-reading files at every plan-phase is painful.

9.4 Tips for large codebases

- Start with `/gsd-scan --focus concerns` to find the rough parts before deciding where to work.
- Run `/gsd-map-codebase` once, then commit `.planning/codebase/` to git. It's a shared asset for the team.
- For repeated work on the same area, enable `/gsd-intel` and refresh it weekly.

- If a phase touches a concern flagged in *CONCERNS.md*, raise the discussion depth — use `--analyze` on `discuss-phase`.

- Run `/gsd-map-codebase` before `/gsd-new-project` on existing code. Every time.
- The four mappers produce four targeted files in `.planning/codebase/`. Commit them.
- `/gsd-scan` is the lightweight alternative. `/gsd-intel` is the queryable alternative.
- Brownfield questions should focus on what you are adding, not what already exists.

Chapter 10

Session Management

GSD is designed to survive interruption. Context window limits, power outages, fires in the middle of a wave, an urgent Slack message — the workflow is structured so that stopping and picking up later is a first-class operation.

10.1 pause-work and HANDOFF.json

`/gsd-pause-work` saves your place. It writes two files:

- `.planning/HANDOFF.json` — a structured machine-readable snapshot: current phase, current wave, executed plans, open questions, git status, open blockers.
- `continue-here.md` — a human-readable summary that Claude (or you) can read at the start of the next session.

You don't have to wait for a context warning to pause. Pause on any natural stopping point — end of a wave, end of a day, after a risky decision you want to sleep on.

10.2 resume-work: what gets restored

`/gsd-resume-work` is the inverse. It reads `HANDOFF.json` and `continue-here.md`, reconstructs the state, and tells you exactly where you were and what's next. Crucially, it does not re-run anything. It only restores context — you still choose the next command.

Run it at the start of any new session. It is the canonical “I'm back, catch me up” command.

10.3 Long-running projects: keeping state clean

Projects that span weeks or months accumulate cruft. GSD's commands help you keep it tidy:

- `/gsd-cleanup` — archives completed phase directories from prior milestones. Keeps *phases/* focused on current work.
- `/gsd-complete-milestone` — archives the milestone into MILESTONES.md and tags it. This is the main pressure valve; use it.
- `/gsd-health` — validates *.planning/* directory integrity. Add `--repair` to auto-fix recoverable issues.
- `/gsd-stats` — dashboard of project metrics. Useful for spotting long-running phases, high token usage, excessive revisions.

Run `/gsd-health --repair` once a week on active projects. It catches STATE.md drift, stale references, and phase directories that got out of sync.

10.4 Context warnings and proactive pausing

GSD ships a context window monitor (`gsd-context-monitor.js`) that runs as a Post-ToolUse hook. It reads metrics from the statusline hook and injects warnings when context usage gets high:

Level	Remaining	Agent behavior
Normal	>35%	No warning.
WARNING	≤35%	Wrap up current task; avoid starting new complex work.
CRITICAL	≤25%	Stop immediately; save state via <code>/gsd-pause-work</code> .

The agent sees the warning in its conversation and can act on it. Enable or disable via `hooks.context_warnings` in `config.json`. Leave it on. The cost is zero; the benefit is “no more running out of context in the middle of a task.”

10.5 Session reports and sharing with humans

`/gsd-session-report` generates a post-session summary with work performed, outcomes, blockers, estimated cost, and next steps. Output lands in *.planning/reports/SESSION_REPORT.*

Useful for:

- Stakeholder updates at the end of a work block.
- “What did I do today?” retrospectives.
- Handing off to a teammate for the next session.

Pair it with `/gsd-milestone-summary` at milestone boundaries for a larger retrospective document.

- `/gsd-pause-work` and `/gsd-resume-work` make interruption safe. Use them liberally.
- `HANDOFF.json` is the structured state; `continue-here.md` is the human summary.
- Context warnings fire at 35% / 25%. Heed them.
- On long-running projects, run `/gsd-health --repair` periodically.

Chapter 11

Backlog, Seeds, and Threads

Ideas show up at inconvenient times. You are deep into Phase 3, and suddenly a perfect idea for Phase 11 arrives. Or for the next milestone. Or for something that should happen “someday, but only if we get WebSocket infrastructure first.” GSD has three different tools for these three different cases.

11.1 The backlog parking lot (999.x numbering)

`/gsd-add-backlog "description"` drops an idea into the backlog. Backlog items are full phase directories, but they are numbered starting at 999.1, 999.2, 999.3, which keeps them outside the active phase sequence. That numbering is deliberate — it means backlog items don’t interfere with `/gsd-next`, and they don’t show up as “phase 11” in the middle of your roadmap.

```
/gsd-add-backlog "GraphQL API layer" # -> 999.1-graphql-api-layer/  
/gsd-add-backlog "Mobile responsive redesign" # ->  
999.2-mobile-responsive-redesign/
```

Because backlog items get real phase directories, you can run `/gsd-discuss-phase 999.1` or `/gsd-plan-phase 999.1` to explore or flesh them out without promoting them yet.

When you are ready to triage, run `/gsd-review-backlog`. It walks you through every backlog item and offers three actions per item:

- **Promote** — move into the active phase sequence. The orchestrator picks the right phase number and slots it in.
- **Keep** — leave it in the backlog for now.
- **Remove** — delete it.

Run `/gsd-review-backlog` before `/gsd-new-milestone`, not in the middle of a phase. It’s a strategy session, not a tactical tool.

11.2 Seeds: forward-looking ideas with triggers

Seeds are different from backlog items. A backlog item is “I want to do this eventually.” A seed is “I want this to surface automatically *when X becomes true.*”

```
| /gsd-plant-seed "Add real-time collaboration when WebSocket infra is in place"
```

This writes `.planning/seeds/SEED-NNN-slug.md`, preserving the full WHY, WHEN, and breadcrumbs to relevant details. Seeds solve context rot: instead of a one-liner in a “Deferred” section that nobody reads, the seed lives as its own file with enough context for a future Claude (or a future you) to pick it up cold.

The trigger happens at `/gsd-new-milestone`. That command scans all seeds and presents matches based on the new milestone’s scope. If you plant a WebSocket-dependent seed and the next milestone adds WebSocket infrastructure, `/gsd-new-milestone` will surface the seed and ask if you want to include it.

11.3 Threads: persistent context across sessions

Threads are the lightest of the three. Use a thread when you have cross-session work that *doesn’t* belong to any specific phase — debugging a production issue, investigating a performance regression, working through a library migration that touches many phases.

```
| /gsd-thread # list all threads
| /gsd-thread fix-deploy-key-auth # resume existing thread
| /gsd-thread "Investigate TCP timeout in pasta service" # create new
```

Each thread file (`.planning/threads/{slug}.md`) has four sections: Goal, Context, References, Next Steps. Threads are lighter than `/gsd-pause-work` — no phase state, no plan context, just a focused cross-session knowledge store.

When a thread matures into a real phase, promote it with `/gsd-add-phase`. When it matures into something for later, promote it with `/gsd-add-backlog`.

11.4 Choosing the right tool

Situation	Use
Idea for eventual work, no strong triggering condition	<code>/gsd-add-backlog</code>
Idea that should surface at a specific future milestone	<code>/gsd-plant-seed</code>
Cross-session investigation that spans phases	<code>/gsd-thread</code>
Tiny capture without structure	<code>/gsd-note</code> or <code>/gsd-add-todo</code>
Full ideation session on a topic	<code>/gsd-explore</code>

- Three tools, three shapes: backlog for deferred work, seeds for triggered surfacing, threads for cross-session investigation.
- Backlog items live as real phase directories at 999.x to stay out of the active sequence.
- Seeds are scanned by `/gsd-new-milestone` and resurface at the right time.
- `/gsd-note` and `/gsd-add-todo` cover smaller captures; `/gsd-explore` covers full ideation.

Chapter 12

Configuration

`.planning/config.json` is the control panel for everything GSD does on your project. Chapter 3 introduced the skeleton. This chapter walks through each group in detail, with the before/after examples you need to actually use them.

12.1 Core settings

Setting	Default	What it controls
mode	interactive	yolo auto-approves decisions; interactive confirms at each gate.
granularity	standard	Phase count: coarse (3-5 phases), standard (5-8), fine (8-12).
model_profile	balanced	Which model tier each agent uses. See Model Profiles below.
project_code	(none)	Prefix for phase directory names (e.g., "ABC" → <code>ABC-01-setup/</code>).
response_language	(none)	Language code for agent responses (e.g., "pt", "ko", "ja").
context_profile	(none)	Pre-configured bundle for specific work types (dev, research, review).

12.2 Workflow toggles

Every workflow toggle follows the *absent equals enabled* rule. If a key is missing, it defaults to `true`. You only write the key to disable the feature.

Setting	Effect when enabled (default)
<code>workflow.research</code>	Run 4 parallel researchers during plan-phase.
<code>workflow.plan_check</code>	Run plan-checker loop (up to 3 iterations).
<code>workflow.verifier</code>	Run verifier after execute-phase.
<code>workflow.nyquist_validation</code>	Map test coverage to requirements during plan-phase.
<code>workflow.ui_phase</code>	Generate UI design contracts for frontend phases.
<code>workflow.ui_safety_gate</code>	Prompt to run ui-phase for frontend work.
<code>workflow.node_repair</code>	Autonomous task repair on verification failure.
<code>workflow.node_repair_budget</code>	Max repair attempts per failed task (default 2).
<code>workflow.auto_advance</code>	Auto-chain discuss → plan → execute without stopping.
<code>workflow.code_review</code>	Enable <code>/gsd-code-review</code> and <code>/gsd-code-review-fix</code> .
<code>workflow.code_review_depth</code>	Default depth: quick / standard / deep.
<code>workflow.discuss_mode</code>	discuss (interview) or assumptions (codebase-first).
<code>workflow.skip_discuss</code>	Bypass discuss-phase in autonomous mode.
<code>workflow.tdd_mode</code>	Enforce RED/GREEN/REFACTOR in executor.
<code>workflow.text_mode</code>	Plain-text menus (for remote Claude Code sessions).
<code>workflow.use_worktrees</code>	Git worktree isolation for parallel execution.

A minimal “prototyping” config that disables quality agents for speed:

```
{
  "mode": "yolo",
  "granularity": "coarse",
  "model_profile": "budget",
  "workflow": {
    "research": false,
    "plan_check": false,
    "verifier": false
  }
}
```

A production config that turns everything up:

```
{
  "mode": "interactive",
  "granularity": "fine",
  "model_profile": "quality",
  "workflow": {
    "code_review_depth": "deep"
  }
}
```

12.3 Model profiles

A profile is a pre-configured bundle of “which model class runs which agent.” GSD ships four:

Agent	quality	balanced	budget	inherit
gsd-planner	Opus	Opus	Sonnet	Inherit
gsd-roadmapper	Opus	Sonnet	Sonnet	Inherit
gsd-executor	Opus	Sonnet	Sonnet	Inherit
gsd-phase-researcher	Opus	Sonnet	Haiku	Inherit
gsd-project-researcher	Opus	Sonnet	Haiku	Inherit
gsd-research-synthesizer	Sonnet	Sonnet	Haiku	Inherit
gsd-debugger	Opus	Sonnet	Sonnet	Inherit
gsd-codebase-mapper	Sonnet	Haiku	Haiku	Inherit
gsd-verifier	Sonnet	Sonnet	Haiku	Inherit
gsd-plan-checker	Sonnet	Sonnet	Haiku	Inherit
gsd-integration-checker	Sonnet	Sonnet	Haiku	Inherit
gsd-nyquist-auditor	Sonnet	Sonnet	Haiku	Inherit

Philosophy:

- **quality** — Opus for everything that makes decisions; Sonnet for read-only verification. Use when quota is available and the work is critical.
- **balanced** — Opus for planning (where architecture decisions live), Sonnet for everything else. The default, for good reason.
- **budget** — Sonnet for anything that writes code, Haiku for research and verification. Good for high-volume or less-critical phases.
- **inherit** — every agent uses the current session model. Use this when switching models dynamically (OpenCode/Kilo /model) or when running Claude Code via non-Anthropic providers (OpenRouter, local models) to avoid unexpected API costs.

You can also override individual agents without changing the whole profile:

```
{
  "model_profile": "balanced",
  "model_overrides": {
    "gsd-executor": "opus",
    "gsd-codebase-mapper": "sonnet"
  }
}
```

Quick profile switch without editing config: `/gsd-set-profile quality` (or balanced, budget, inherit).

12.4 Git branching

Strategy	Creates branch	Best for
none (default)	Never	Solo development, simple projects.
phase	At each <code>/gsd-execute-phase</code>	Code review per phase, granular rollback.
milestone	At first <code>/gsd-execute-phase</code> of the milestone	Release branches, PR per version.

Templates control the branch names:

```
"git": {
  "branching_strategy": "phase",
  "phase_branch_template": "gsd/phase-{phase}-{slug}",
  "milestone_branch_template": "gsd/{milestone}-{slug}",
  "quick_branch_template": "gsd/quick-{num}-{slug}"
}
```

Template variables: `{phase}` (zero-padded), `{slug}` (hyphenated name), `{milestone}` (version), `{num}` (quick task ID).

12.5 Parallelization

Setting	Default	Effect
<code>parallelization.enabled</code>	<code>true</code>	Run independent plans in parallel.
<code>parallelization.max_concurrent_agents</code>	<code>3</code>	Maximum simultaneous executors.
<code>parallelization.min_plans_for_parallel</code>	<code>2</code>	Below this, force sequential.
<code>parallelization.skip_checkpoints</code>	<code>true</code>	Skip checkpoint gates during parallel runs.

If you are on Rust or any language with build-lock contention and you're seeing cargo lock fights, this is the subsystem to check. GSD since v1.26 commits parallel-wave executors with `--no-verify` and runs hooks once per wave, which fixes most issues. If you need hooks per commit for compliance, set `parallelization.enabled: false` and take the sequential hit.

12.6 Hooks and security

Setting	Default	Effect
<code>hooks.context_warnings</code>	<code>true</code>	Context window usage warnings.
<code>hooks.workflow_guard</code>	<code>false</code>	Warn on edits outside GSD workflow context.
<code>security_enforcement</code>	<code>true</code>	Enable threat-model-anchored verification via <code>/gsd-secure-phase</code> .
<code>security_asvs_level</code>	<code>1</code>	OWASP ASVS verification level (1 / 2 / 3).
<code>security_block_on</code>	<code>high</code>	Minimum severity that blocks phase advancement.

The prompt-injection guard hook is always active and cannot be disabled. It scans Write/Edit calls to `.planning/` for known injection patterns (role overrides, instruction bypasses, system tag injection) in advisory-only mode. See Appendix A for the full security context.

12.7 Interactive configuration

You do not have to edit `config.json` by hand. `/gsd-settings` walks you through every option interactively. `/gsd-set-profile` flips the model profile in one command. Behind the scenes both tools edit the same file.

- `absent` equals `enabled` — missing workflow keys default to on.
- `balanced` is the default profile for a reason; only change it deliberately.
- `parallelization.enabled: false` is the escape hatch for build-lock contention.
- Use `/gsd-settings` when you don't want to touch JSON.

Chapter 13

UI Design Workflow

The default AI-generated frontend looks mostly fine but never quite right: spacing is inconsistent, typography drifts, the fifth button doesn't match the first four. This is not a model failure. It is the absence of a design contract. Five components built without a shared spacing scale, color scheme, or copywriting standard produce five slightly different visual decisions. GSD's UI commands fix that by locking the contract before planning and auditing the result after execution.

13.1 ui-phase: lock the contract

Run `/gsd-ui-phase N` after `/gsd-discuss-phase` and before `/gsd-plan-phase`, for any phase that does frontend work. The flow:

1. Read `CONTEXT.md`, `RESEARCH.md`, and `REQUIREMENTS.md` for existing decisions.
2. Detect the current design-system state: `shadcn components.json`, Tailwind config, existing tokens.
3. If the project is React/Next.js/Vite and no `shadcn` is installed, offer a **shadcn initialization gate** (more on this below).
4. Ask only unanswered design-contract questions (spacing, typography, color, copywriting, registry safety).
5. Write `{phase}-UI-SPEC.md`.
6. Validate against the 6 pillars; revision loop if blocked (max 2 iterations).

13.2 The 6 pillars

Every `UI-SPEC.md` and every `/gsd-ui-review` scores against six dimensions:

Pillar	What it covers
Copywriting	CTA labels, empty states, error states, microcopy consistency.
Visuals	Focal points, visual hierarchy, icon accessibility.
Color	Accent usage discipline, 60/30/10 compliance.
Typography	Font size and weight constraint adherence.
Spacing	Grid alignment, token consistency.
Experience Design	Loading / error / empty state coverage.

Each pillar is scored 1–4 after execution. A phase that scores 4/4 on all six pillars is visually consistent. Anything below 3 on any pillar is a real finding.

13.3 shadcn initialization gate

For React/Next.js/Vite projects without shadcn, the UI researcher offers to initialize it. The flow:

1. Visit `ui.shadcn.com/create` and configure your preset (colors, radius, fonts).
2. Copy the preset string.
3. Run `npx shadcn init --preset {paste}`.
4. The preset string becomes a first-class GSD planning artifact, reproducible across phases and milestones.

The preset encodes the entire design system in one string. Commit it.

13.4 Registry safety gate

Third-party shadcn registries can inject arbitrary code. The safety gate — enabled via `workflow.ui_safety_gate: true` (the default) — requires two inspection steps before installing any registry component:

- `npx shadcn view {component}` — inspect before installing.
- `npx shadcn diff {component}` — compare against the official version.

This is not paranoia. Registry components are arbitrary code that runs in your project. The gate is a pause-and-look step, nothing more. Leave it on.

13.5 ui-review: retroactive visual audit

`/gsd-ui-review N` is the retroactive side of the workflow. Run it after execution or after `/gsd-verify-work`. It works on any project with frontend code, not just GSD-managed ones. If no UI-SPEC.md exists, it audits against the abstract 6-pillar standards.

The review captures screenshots via Playwright CLI to `.planning/ui-reviews/` (auto-gitignored), scores each pillar 1–4, and writes `{phase}-UI-REVIEW.md` with the scores and a top-3 priority fix list.

13.6 The ui-phase safety gate

When you run `/gsd-plan-phase` on a phase with frontend work and no `UI-SPEC.md` exists, `workflow.ui_safety_gate: true` causes plan-phase to pause and prompt you: “This phase looks like frontend work. Run `/gsd-ui-phase` first?” You can say no and continue planning anyway, but the default is to stop you before you plan frontend work without a design contract. Disable by setting `workflow.ui_safety_gate: false`.

- `/gsd-ui-phase` locks the design contract *before* planning; `/gsd-ui-review` audits it after.
- Six pillars: Copywriting, Visuals, Color, Typography, Spacing, Experience Design.
- The `shadcn` preset string is a first-class planning artifact. Commit it.
- The registry safety gate is a pause-and-look step, not a blocker. Keep it on.

Chapter 14

Troubleshooting

Even with good practices, things go wrong. This chapter is a symptom-to-remedy catalog — when X happens, run Y. It is organized by what you observe, not by command, so you can find the fix without already knowing which command to reach for.

14.1 ``Commands not found''

Symptom: you type `/gsd-new-project` and Claude says it doesn't recognize the command. Diagnosis: GSD is not installed for the runtime you are using.

Fixes:

- Re-run `npx get-shit-done-cc@latest`.
- For a non-Claude runtime, make sure you installed with the right flag: `--gemini`, `--opencode`, `--codex`, `--kilo`, `--copilot`, `--cline`, `--qwen`, `--codebuddy`.
- For local-only install, use `--local`. For global, use `--global` (or omit for default global).
- Check the version: `/gsd-update` shows what's installed and what's available.

14.2 ``Plans keep failing verification''

Symptom: `/gsd-plan-phase` produces plans that the plan-checker rejects repeatedly and eventually surfaces as blocked. Diagnosis: almost always a discussion gap. The planner is filling in decisions that `CONTEXT.md` should have locked.

Fixes:

- Re-run `/gsd-discuss-phase N`, this time with `--analyze`. Go deeper. Give the planner enough to work with.
- Check the phase goal in `ROADMAP.md`. If it's vague, tighten it and re-plan.

- If the phase covers too much, split it with `/gsd-insert-phase` before re-planning.
- Run `/gsd-list-phase-assumptions N` to preview what Claude intends to do before committing.

14.3 ``Execution fails or produces stubs''

Symptom: executor finishes but code is half-built, tests are stubs, or whole features are missing. Diagnosis: the plan had too much in one task, or `CONTEXT.md` was vague, or the task's acceptance criteria were weak.

Fixes:

- Read `PLAN.md`. If any task has more than 2-3 concrete actions, the task is too big for one context. Re-plan with finer granularity.
- Check `CONTEXT.md`. If the specifics section is one-liners, re-discuss.
- Re-plan with `--skip-research` and a tighter `CONTEXT.md` — the problem isn't research, it's intent.

14.4 ``Plans seem wrong or misaligned''

Symptom: the plan solves the wrong problem, chooses libraries you don't want, or misses an obvious consideration. Diagnosis: `CONTEXT.md` didn't capture what you meant.

Fix: re-run `/gsd-discuss-phase N` — most plan-misalignment issues trace back to assumptions `CONTEXT.md` would have prevented. Alternatively, `/gsd-list-phase-assumptions N` shows you what Claude intends before you commit.

14.5 ``Discuss-phase uses technical jargon I don't understand''

Symptom: the question set is full of terms you haven't seen before. Fix: `/gsd-profile-user` to generate a behavioral profile. If the profile indicates a non-technical owner — `learning_style: guided`, jargon listed as a frustration trigger, or `explanation_depth: high-level` — gray-area questions are automatically reframed in product-outcome language instead of implementation terminology. The profile is stored at `~/.claude/get-shit-done/US` and is read automatically on every `/gsd-discuss-phase` invocation.

14.6 ``STATE.md is out of sync''

Symptom: `/gsd-progress` reports the wrong phase, or `/gsd-next` suggests the wrong command. Diagnosis: `STATE.md` drifted from filesystem reality (a rebase, a manual edit, a crash mid-write).

Fixes:

- `node gsd-tools.cjs state validate` — detects the drift.
- `node gsd-tools.cjs state sync --verify` — dry-run showing what would change.
- `node gsd-tools.cjs state sync` — reconstructs STATE.md from disk.

These commands replaced manual STATE.md editing in v1.32. Use them.

14.7 “I don't know what went wrong” --- gsd-forensics

When state, artifacts, or git history seem corrupted and you can't even describe the problem, run `/gsd-forensics`. It generates a diagnostic report covering:

- Git history anomalies (orphaned commits, unexpected branch state, rebase artifacts, time gaps).
- Artifact integrity (expected files for completed phases, broken cross-references).
- STATE.md anomalies and session history.
- Uncommitted work, conflicts, abandoned changes.
- At least 4 anomaly types checked: stuck loop, missing artifacts, abandoned work, crash/interruption.

Output lands in `.planning/forensics/report-{timestamp}.md`. Optionally offers to file a GitHub issue if findings are actionable. This is the “I'm stuck, tell me what happened” command.

14.8 gsd-health repair

`/gsd-health` runs a series of integrity checks on `.planning/`: missing files, broken cross-references, STATE.md drift, phase directory consistency. Add `--repair` to auto-fix recoverable issues.

Run this weekly on active projects. It catches small issues before they become forensics-level problems.

14.9 Systematic debugging with gsd-debug

For actual bugs in your code (not in GSD itself), `/gsd-debug` is the structured path. It maintains persistent debug-session state so you can investigate, abandon, and resume without losing hypothesis history.

Subcommands:

- `/gsd-debug ``description''` — start a new session.
- `/gsd-debug --diagnose ``description''` — investigate without attempting fixes.
- `/gsd-debug list` — list all active sessions.

- `/gsd-debug status <slug>` — full summary of one session.
- `/gsd-debug continue <slug>` — resume a specific session.

When `workflow.tdd_mode: true`, debug sessions require a failing test before any fix is applied (red → green → done).

14.10 Context degradation in long sessions

Symptom: quality drops mid-session, Claude starts contradicting itself, past decisions get ignored. Diagnosis: you are in context rot.

Fix: `/clear` in Claude Code to start a fresh session. Then `/gsd-resume-work` to restore project context from `HANDOFF.json` and `continue-here.md`. GSD is designed around fresh contexts — every subagent already gets a clean 200K window. The top-level session is the one that rots, and `/clear` is the cure.

14.11 Parallel execution causes build lock errors

Symptom: pre-commit hook failures, cargo lock contention, 30+ minute execution times. Diagnosis: parallel executors triggering build tools simultaneously. Fix on v1.26+: nothing — GSD handles it. On older versions: add this to your project's `CLAUDE.md`:

```
## Git Commit Rules for Agents
All subagent/executor commits MUST use `--no-verify`.
```

To disable parallel execution entirely (last resort): `/gsd-settings` → set `parallelization.enabled` to `false`.

14.12 Quick recovery reference

Problem	Solution
Lost context / new session	<code>/gsd-resume-work</code> or <code>/gsd-progress</code>
Phase went wrong	<code>/gsd-undo --phase NN</code>
Need to change scope	<code>/gsd-add-phase</code> , <code>/gsd-insert-phase</code> , or <code>/gsd-remove-phase</code>
Milestone audit found gaps	<code>/gsd-plan-milestone-gaps</code>
Something broke	<code>/gsd-debug ``description``</code>
STATE.md out of sync	<code>state validate</code> then <code>state sync</code>
Workflow state seems corrupted	<code>/gsd-forensics</code>
Quick targeted fix	<code>/gsd-quick</code>
Plan doesn't match your vision	<code>/gsd-discuss-phase N</code> then re-plan
Costs running high	<code>/gsd-set-profile budget + /gsd-settings</code>
Update broke local changes	<code>/gsd-reapply-patches</code>
Don't know what step is next	<code>/gsd-next</code>

- Most plan failures are discussion failures. Re-run `/gsd-discuss-phase` first.
- Use `state validate` and `state sync` instead of hand-editing STATE.md.
- `/gsd-forensics` is the “I’m stuck” command. Run it before giving up.
- `/gsd-health --repair` weekly keeps `.planning/` clean.

Appendix A

Permissions Deep Dive

GSD is designed for frictionless automation. The maintainers recommend running Claude Code with `--dangerously-skip-permissions`, and say so in the README. This appendix lays out what that flag actually does, what the granular alternative looks like, and which approach is appropriate for which situation. Both are real and supported — pick with eyes open.

A.1 What `--dangerously-skip-permissions` does

This flag, when passed to Claude Code, skips the per-tool-call confirmation prompts that Claude Code normally shows. Without the flag, every shell command (`ls`, `grep`, `git status`, everything) requires a `yes-from-you`. With the flag, Claude runs commands directly.

The honest case for it: GSD generates dozens of trivial shell calls per phase — `date`, `git add`, `git commit`, `cat`, `ls`, `grep`. Stopping to approve each one is friction that defeats the point of automated planning. On a full phase with 15 tasks, you could easily be approving 200+ commands.

The honest case against it: the flag is called “dangerously” for a reason. It removes the single biggest safety net that Claude Code offers. An agent with this flag on can execute any command it decides to — including destructive ones — without asking you. If a prompt-injection attack succeeded in steering the agent, the blast radius is the blast radius of shell access.

`--dangerously-skip-permissions` disables Claude Code’s primary shell-call safety gate. Use it in trusted project directories you control, not in arbitrary checkouts of code you haven’t reviewed. Never use it in a session that is about to touch production infrastructure, credentials, or anything you can’t `git reset` your way out of.

A.2 The granular alternative

If `--dangerously-skip-permissions` is too much, you can pre-approve a specific allow list instead. Add this to your project's `.claude/settings.json` (the file Claude Code reads for per-project configuration):

```
{
  "permissions": {
    "allow": [
      "Bash(date:*)",
      "Bash(echo:*)",
      "Bash(cat:*)",
      "Bash(ls:*)",
      "Bash(mkdir:*)",
      "Bash(wc:*)",
      "Bash(head:*)",
      "Bash(tail:*)",
      "Bash(sort:*)",
      "Bash(grep:*)",
      "Bash(tr:*)",
      "Bash(git add:*)",
      "Bash(git commit:*)",
      "Bash(git status:*)",
      "Bash(git log:*)",
      "Bash(git diff:*)",
      "Bash(git tag:*)"
    ]
  }
}
```

This pattern whitelists the specific commands GSD needs to run without prompting, while keeping everything else gated. If Claude tries to run `rm -rf`, `curl pipe-to-shell`, or anything not in the allow list, Claude Code will still stop and ask you.

The allow list is scoped to this project's `.claude/settings.json`. It does not affect other projects. The pattern syntax is Claude Code's, not GSD's — `Bash(command:*)` means "any invocation of command."

A.3 Recommended allow lists by workflow

Different GSD workflows need different tool sets. Start with the base list above and add as needed.

Base (always needed): as shown above — filesystem basics, git read/write, standard Unix text tools.

Testing: add `Bash(npm test:*)`, `Bash(pytest:*)`, `Bash(cargo test:*)`, `Bash(go test:*)`, `Bash(jest:*)`, etc., depending on your stack.

Building: add `Bash(npm run build:*)`, `Bash(cargo build:*)`, `Bash(go build:*)`, `Bash(tsc:*)`.

Package management: add `Bash(npm install:*)`, `Bash(cargo add:*)`, `Bash(pip`

install:*) — but *only* if you trust GSD to pick dependencies in this project. Many teams deliberately keep these gated.

Shipping (optional): add `Bash(gh pr create:*)`, `Bash(gh pr view:*)`, `Bash(gh issue create:*)` if you use `/gsd-ship`.

Node tooling for GSD: add `Bash(node gsd-tools.cjs:*)` if you run state commands manually.

Do *not* blanket-allow `Bash(*)`. That is equivalent to `--dangerously-skip-permissions` with more typing.

A.4 Choosing between the two

Situation	Recommendation
Personal side project, local dev, throwaway repo	--dangerously-skip-is reasonable.
Team project, shared codebase, bounded trust	Granular allow list.
Production-adjacent work (touches deployed configs, credentials, release tags)	Granular allow list, minimal scope.
Learning / exploring GSD for the first time	Start with granular; you'll learn which commands matter.
CI / headless automation	Use the granular allow list via project settings, not the flag.
Brand new runtime you haven't vetted	Granular only, until you've

A.5 What GSD does on its own

Separately from the permissions flag, GSD ships two runtime hooks that provide defense in depth. These are not opt-in:

- **gsd-prompt-guard.js** — PreToolUse hook that scans Write/Edit calls targeting `.planning/` for known injection patterns (role overrides, instruction bypasses, system-tag injection). Advisory-only — it flags but does not block. Always active, cannot be disabled.
- **gsd-workflow-guard.js** — warns when file edits happen outside a GSD workflow context (advises using `/gsd-quick` or `/gsd-fast`). Opt-in via `hooks.workflow_guard: true`. Off by default.

These protect `.planning/` specifically because planning files become system prompts for future agents. User-supplied text flowing into planning artifacts is a potential indirect-prompt-injection vector, and the guard is the defense-in-depth layer for it. A prompt-injection scanner (`prompt-injection-scan.test.cjs`) also runs in GSD's CI against all agent, workflow, and command files.

A.6 Summary

`--dangerously-skip-permissions` is real, recommended for personal projects, and removes every per-command safety gate. The granular allow list gives you the same automation flow with an explicit trust boundary. For team, production-adjacent, or shared work, the granular approach is the right default. For personal projects in directories you control, the flag is honest and fine. Both are supported. Pick with eyes open, and re-evaluate when your context changes.

- `--dangerously-skip-permissions` removes all per-command gates. Fine for personal work in directories you control.
- The granular `permissions.allow` list is the right default for team, shared, or production-adjacent work.
- GSD's prompt-guard hook protects `.planning/` unconditionally and cannot be disabled.
- Never blanket-allow `Bash(*)`. If you want that, just use the flag.