n'co studio

R&D lab for markpreston.co

REPO OS TEMPLATE · EXPLAINER

How Prompt Engineering Works in the Repo OS Template

The operator supplies direction. The system supplies structure, review, and enforcement.

AUTHOR	DATE	SCOPE	TYPE
Mark Preston	2026-07-06	Repo OS template	Explainer

SECTION 00

The one-sentence version

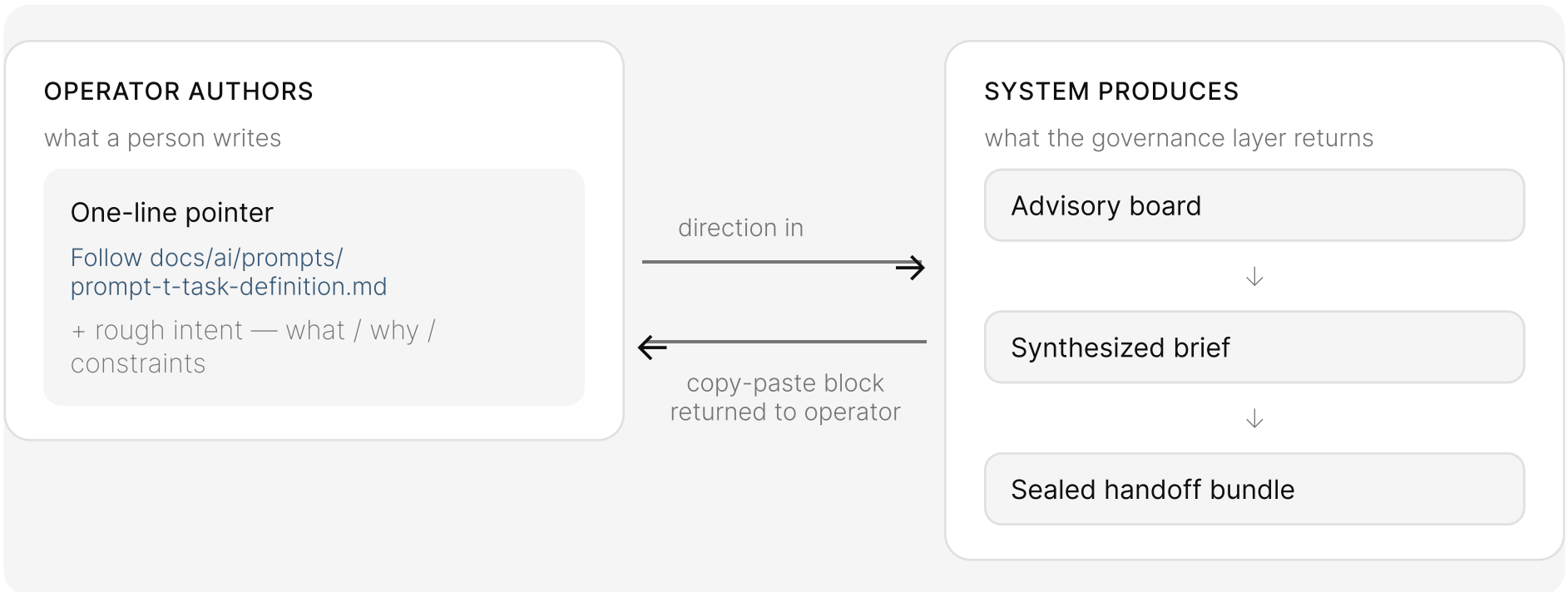
What “prompt engineering” means inside Repo OS — and what it doesn’t.

Not prose in a box

In Repo OS, “prompt engineering” does not mean writing clever prose into a chat box. It means invoking one of a fixed library of governed, versioned prompt files — each a facilitator persona with a defined phase sequence — and letting the governance layer (specs, an advisory board of AI specialists, hooks, and decisions) turn a rough intent into an approved, machine-readable brief that a build session can execute deterministically.

> **IN ONE LINE**
> The operator supplies direction. The system supplies structure, review, and enforcement.

DIAGRAM 1 · THE TWO HALVES



The operator writes a pointer and a rough intent; the governance layer returns a synthesized brief and a sealed handoff bundle to copy-paste into the next session.

SECTION 01

Prompts are pointers, not paragraphs

The mental model — a session starts from a one-line pointer, not a wall of prose.

A session starts with one line

A Repo OS session begins when the operator pastes a one-line pointer into a fresh session. That single line is the "prompt." Everything else — persona, rules, phase sequence, which files to read, which hooks fire — lives inside the referenced .md file and the governance index it loads (AI.md).

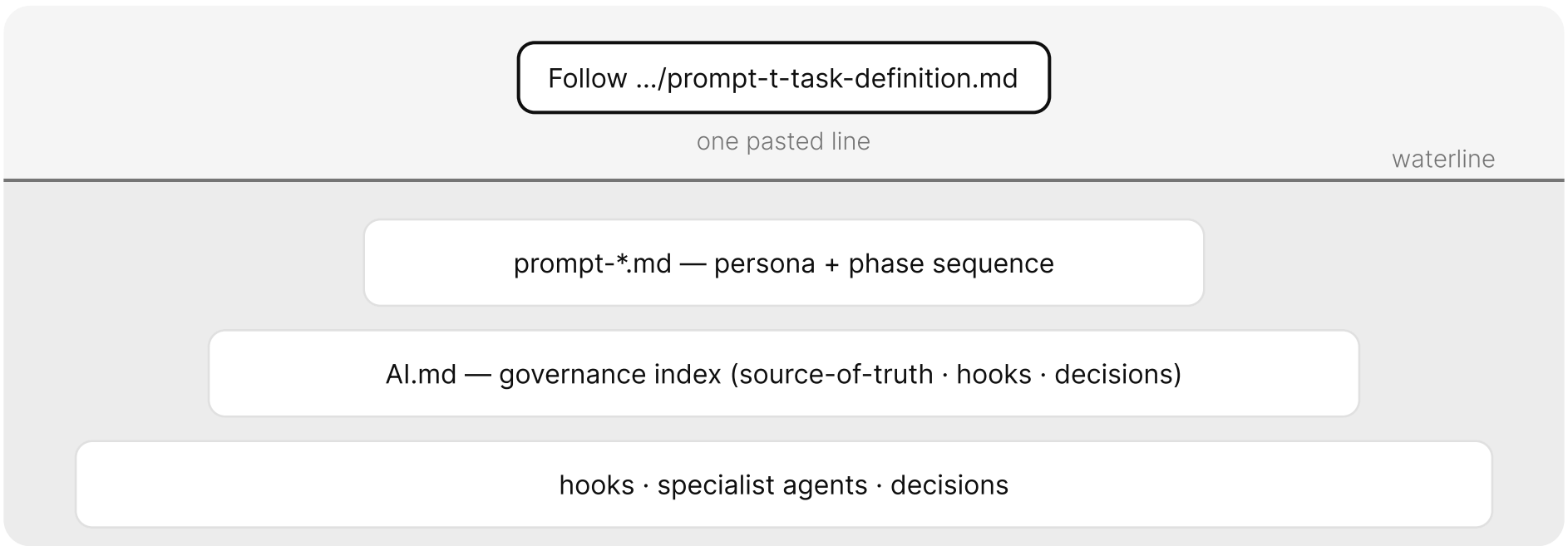
> **THE POINTER**
>
> Follow docs/ai/prompts/prompt-t-task-definition.md. Start in Query mode.

Why this matters

Simple version: the operator never has to remember the rules. The rules travel with the prompt file, and the prompt file is version-controlled (CalVer git tags). Upgrade the template, and every future prompt automatically inherits the new discipline.

Technical version: each prompt file is a self-contained contract that reads AI.md and its source-of-truth references, calls init_phase via Bash to register the session with the hook layer, declares a phase sequence enforced by _phase-helper.sh (Decision 0014), and defines which specialist agents may be dispatched and in which mode.

DIAGRAM 2 · ANATOMY OF AN INVOCATION



The pasted line is the tip; the persona, the governance index, and the enforcement layer all sit below the surface.

SECTION 02

The prompt library

The “keyboard” the operator plays — about eleven canonical entry-point prompts, each a distinct mode of work.

One pointer per mode

Repo OS ships roughly eleven canonical entry-point prompts in docs/ai/prompts/. Each is a distinct mode of work with its own phase sequence. The operator picks one by pasting its pointer — the table below is the whole instrument.

Prompt	Name	What the operator uses it for	Writes code?
0	Spec Builder	Interview to build the project spec before any code	No — specs only
0.5	Rapid Prototype	Lightweight docs for experiment/* spikes	Yes (A4 gate only)
A	Repo OS Init & Advisory Review	Initialize / validate the governance model; run the Advisory Board over specs	No product code
T	Task Definition	The brief factory — explore (Query) or produce a build brief (Brief)	No (writes bundle only)
B	Product Build	Execute an approved brief through the gates to merge	Yes
C	Project Audit	Health audit across dimensions (drift, compliance, tech-debt…)	No
E	Editor Live	React to live external-editor saves; build + commit each change	Yes
F	Finish Loop	Iterative assess → fix → reassess to close gaps	Yes
H	Hotfix & Debug	Reactive debugging on hotfix/*, expedited gates	Yes
Q	Queue Triage	Conversationally manage the headless automation queue	No
D	Daemon Toggle	Turn the automation runner on / off	Config only

THE ELEVEN PROMPTS, AS A BOARD

0 Spec Builder governance / planning	0.5 Rapid Prototype writes code	A Init & Advisory governance / planning
T Task Definition governance / planning	B Product Build writes code	C Project Audit governance / planning
E Editor Live writes code	F Finish Loop writes code	H Hotfix & Debug writes code
Q Queue Triage governance / planning	D Daemon Toggle governance / planning	

warm — writes code cool — governance / planning only

SECTION 03

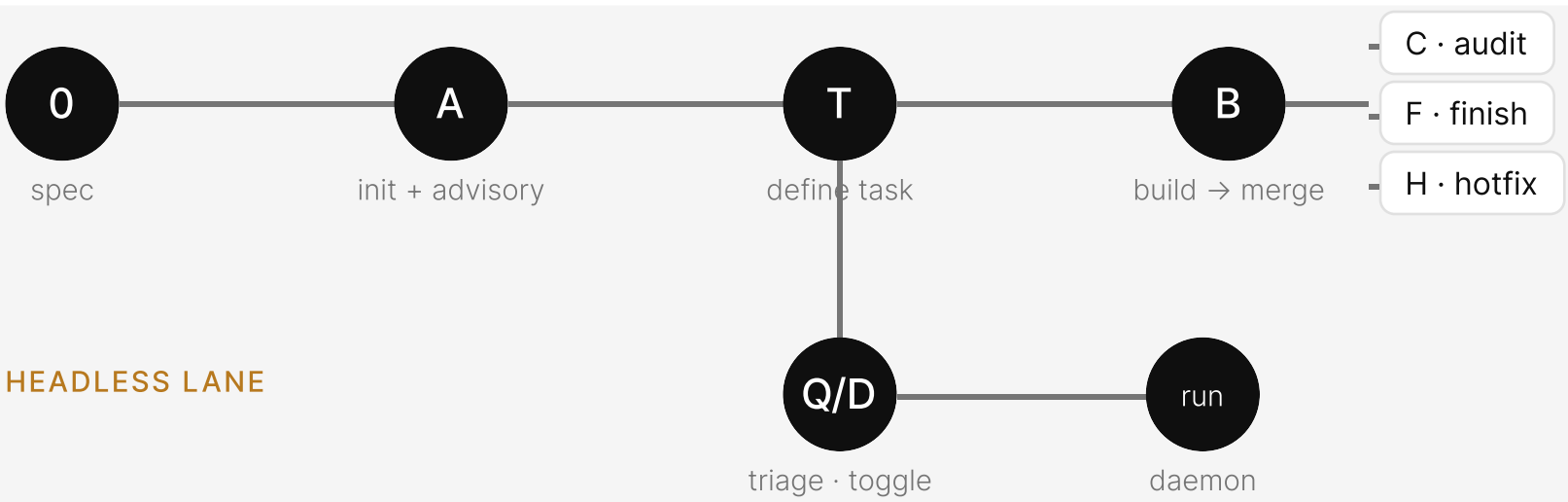
How prompts chain into a pipeline

Prompts are not used in isolation — they form a lineage from empty repo to shipped feature.

A lineage, not a grab-bag

The canonical path runs 0 (spec) → A (init + advisory review) → T (define task → approved brief) → B (build → gates → merge), with C / F / H branching off the build for audit, finish, and hotfix. Prompt 0.5 branches off for throwaway experiments; Prompt Q / D manage the headless automation lane.

DIAGRAM 4 · PROJECT LIFECYCLE RAIL



Main line 0 → A → T → B; audit / finish / hotfix branch off the build, and a parallel headless lane runs T → Q / D → daemon.

SECTION 04

Prompt T — the brief factory

Where the governance model produces prompts for the operator. Two modes, selected by the operator's language.

QUERY MODE

Conversational exploration. The Orchestrator answers directly from compiled specs and the codebase. No agent dispatch, no files written. (This explainer was produced in Query mode.)

BRIEF MODE

Full task definition. Dispatches an advisory board, synthesizes a brief, gets operator approval, and emits a copy-pasteable brief plus a sealed handoff bundle.

BRIEF MODE, STEP BY STEP

1. **Gather context**
   The operator gives What / Why / Constraints. T runs fix-on-fix detection (is this the Nth patch to the same system?) and asks about User Review when the task is user-facing.

2. **Dispatch specialists in parallel**
   The Advisory Board runs concurrently. Each specialist gets a pre-assembled inline context payload; selective spec loading (Decision 0035) loads only task-relevant spec files, not the whole directory.

3. **Synthesize**
   The Orchestrator folds each specialist's findings into brief sections — Architect → Scope/Approach, Critic → Risks/Verification, Pragmatist → right-sizing, PM → Constraints, UX → design brief. A pipeline-scoping heuristic forces end-to-end coverage.

4. **Operator review**
   T presents a summary and asks for approval via AskUserQuestion — a structured dialog, never plain text, because it triggers the plan-approval.sh hook. Options: Approve — review / auto, Approve & queue, Revise, Abandon, Narrow / Split.

5. **Output**
   T writes the handoff bundle and prints the finalized brief as a self-contained fenced block the operator copies into a new session.

§4 · PROMPT T — BRIEF MODE FLOW

From a rough intent to a sealed, copy-paste brief

OPERATOR

Gather context

What / Why / Constraints. T runs fix-on-fix detection and asks about User Review if the task is user-facing.

↓

ORCHESTRATOR

Dispatch advisory board

A panel of read-only specialists runs in parallel, each with a pre-assembled inline context payload (Decision 0035 loads only task-relevant specs).

↓

Architect	Exploration	Critic	Pragmatist	PM	UX
core	core	core	core	cond.	cond.

↓

ORCHESTRATOR

Synthesize

Fold each specialist's findings into brief sections. A pipeline-scoping heuristic forces end-to-end coverage rather than splitting a cross-stage change.

↓

OPERATOR GATE

AskUserQuestion — approval

A structured dialog (never plain text) that fires the plan-approval.sh hook. Revise loops back to Synthesize; Approve & queue peels off to the daemon lane.

↓

ORCHESTRATOR

Write bundle + print brief

T writes the sealed handoff bundle and prints the finalized brief as a copy-paste fenced block for a new session.

§4.2 · THE THREE BRIEF SHAPES

What Prompt T emits depends on the size of the task

A single change becomes one brief. A change that needs spec-level definition becomes a sequential pair. A change that is too big to hold becomes a parallel group with an auto-generated merge brief.

Single brief the standard case	brief → Prompt B
Two-brief task group new model · integration · boundary	Brief 1 · spec → Brief 2 · build
Narrow / split operator says “too big”	B1 · worktree B2 · worktree Bn · worktree → Brief M · merge

PARALLEL EXECUTION

When sub-brief file scopes don’t overlap, T offers parallel execution — each Bn in its own git worktree (claude --worktree, unique REPO_OS_SESSION_ID) — plus an auto-generated Brief M that combines the branches and runs a cross-session cohesion review before merge.

SECTION 05

The Advisory Board

The AI specialist panel behind every brief — read-only agents dispatched in parallel, each reviewing the task from one angle.

Findings are advisory — the operator decides

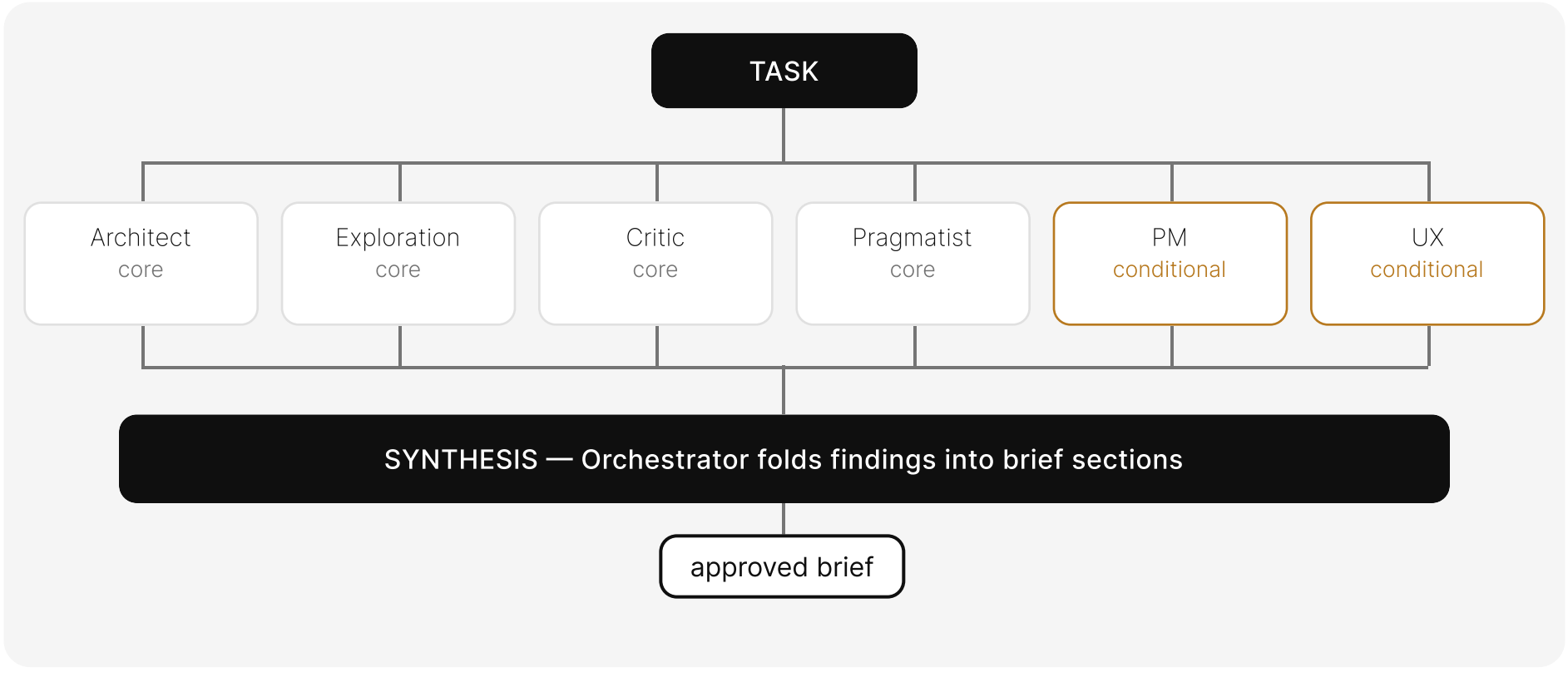
Like handing a rough idea to a small expert panel — an architect, a skeptic, an explorer, a minimalist, a PM, and a designer — who all read it at once and hand back notes, which the Orchestrator staples into a coherent plan. All run concurrently, receive an inline context payload so they don't re-read shared files, and are write-blocked by disallowedTools + pretooluse-permission.sh.

Specialist	Angle it reviews	Status
Architect	Where the change fits — which modules, structural patterns, ordering. Emits an Implementation Design block when scope exceeds Small.	Core
Exploration	Alternative approaches, libraries, best practices.	Core
Critic	What could go wrong — gaps, unstated assumptions, contradictions, risks.	Core
Pragmatist	Is it proportionate? Over-engineering, scope inflation. Every finding carries a falsifiability condition.	Core
PM Advisory	MVP-scope fit, spec consistency, sequencing.	Conditional · spec exists
UX Design Agent	Layout, hierarchy, interaction, accessibility — emits a design brief.	Conditional · UI mentioned

SEVERITY SCALE (SHARED)

CRITICAL — address before build · RECOMMENDATION — operator decides · IDEA — nice-to-have. PM adds orthogonal scope labels: IN-SCOPE / OUT-OF-SCOPE / DEFER.

DIAGRAM 7 · ADVISORY BOARD — FAN-OUT / FAN-IN



Each finding carries a severity — CRITICAL · RECOMMENDATION · IDEA — and PM scope labels IN-SCOPE / OUT-OF-SCOPE / DEFER.

SECTION 06

The handoff bundle

How an approved brief becomes executable — the brief the operator copies is backed by a sealed, machine-readable artifact.

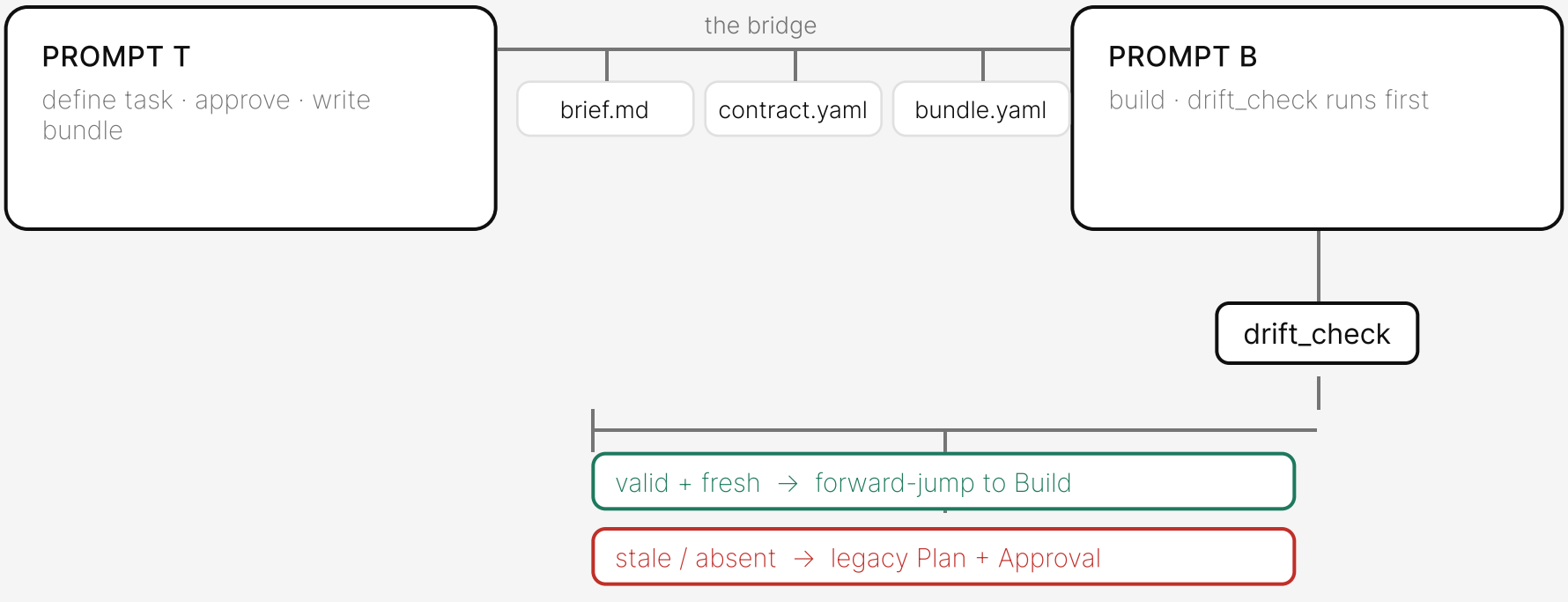
Three things written on approval

On any Approve path, Prompt T writes three things: the brief (human-readable markdown, pasted into a new Prompt B session); the Task Contract (a YAML block embedded in the brief — brief_id, goal, scope_files, approach, verification, spec_references, constraints — the machine-readable spine); and the handoff bundle, written atomically to .claude/local/handoff-bundles/<brief_id>.yaml (per-brief, collision-safe — Decision t-0043), carrying the contract, plan, and a spec_snapshot (sha256 of each referenced spec file at approval time).

THE PAYOFF — DRIFT_CHECK (DECISION 0038)

When the operator pastes the brief into a fresh Prompt B session, B's first act is drift_check. Bundle valid + fresh → forward-jump straight to build, skipping the plan and approval gates (approval already happened in T). Bundle absent / stale / spec drifted → fall through to the legacy plan → approval path.

DIAGRAM 8 · THE HANDOFF-BUNDLE BRIDGE



The spec_snapshot (sha256 of each referenced spec at approval time) is exactly what drift_check re-checks.

SECTION 07

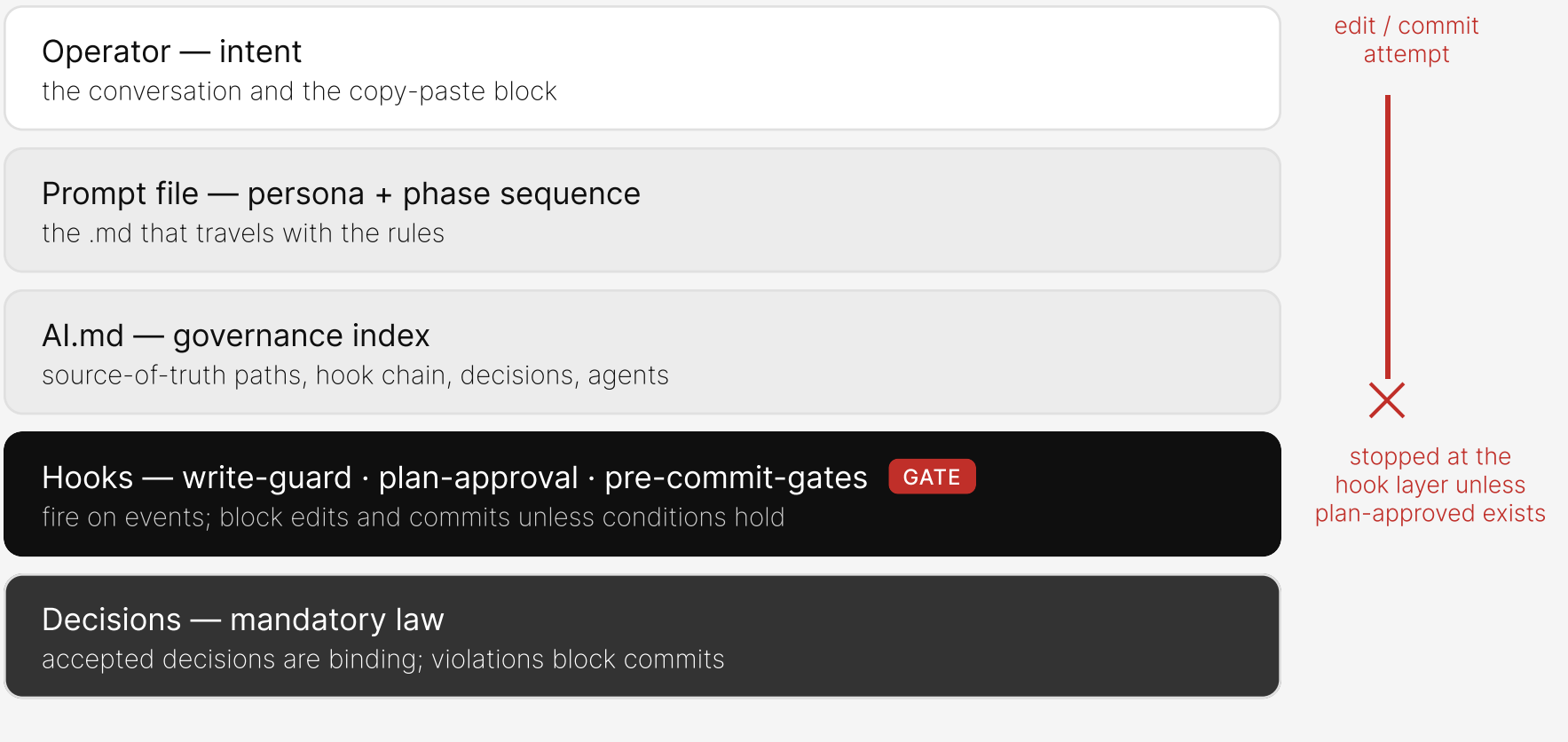
How governance supports the operator

The operator sees a conversation and a copy-paste block. Underneath, four layers make that possible — and safe.

The invisible scaffolding

The operator drives; the guardrails are welded to the road. You literally cannot commit code without an approved plan and passing gates — not by policy, by mechanism. Four layers do the work: AI.md (the index every prompt reads first), Decisions (binding law — violations block commits), Hooks (automatic enforcement Claude Code fires on events), and the phase model (every session type declares an ordered phase sequence, and hooks are phase-aware).

DIAGRAM 9 · GOVERNANCE STACK — ENFORCEMENT CROSS-SECTION



Enforcement is mechanical, not advisory — the hook layer stamps GATE on any edit or commit that lacks an approved plan and passing gates.

THE KEY HOOKS

write-guard.sh / bash-guard.sh	block edits until a plan-approved state file exists; enforce branch isolation and main-branch protection
plan-approval.sh	writes plan-approved when the operator confirms via AskUserQuestion — or from a valid handoff bundle
pre-commit-gates.sh	runs gates A1–A6 (scope, secret scan, cohesion + diff-review, contract existence, functional verification) before any commit
pretooluse-permission.sh	session-type-aware agent allowlist; blocks write-capable subagents in no-write sessions
session-close-check.sh	session-log + merge safety net at close

THE PHASE MODEL (DECISION 0014)

Query: init → conversation → close. Brief: init → dispatch → synthesize → approval → output → close. Build: init → drift-check → [plan → approval] → build → verify → commit → close.

SECTION 08

End-to-end operator workflows

The copy-paste choreography — how a user actually works.

DIAGRAM 10 · WORKFLOW A — EXPLORE, THEN BUILD (THE COMMON CASE)

1 PROMPT T

Paste the Query-mode pointer into a fresh session.

2 PROMPT T

Discuss feasibility conversationally. No files written.

3 PROMPT T

Say “turn this into a brief” → T re-inits to Brief mode.

4 PROMPT T

T dispatches the board, synthesizes, asks approval via AskUserQuestion.

5 PROMPT T

Choose Approve — review (or auto).

6 PROMPT T

T prints a copy-paste brief block + writes the handoff bundle.

7 PROMPT B

Open a NEW session, paste the brief (points to prompt-b-product-build.md).

8 PROMPT B

B drift-checks the bundle → Build → gates → merge to main.

The artifact handed across the session boundary is the copy-paste brief block plus its sealed bundle (steps 6 → 7).

DIAGRAM 11 · FOUR WORKFLOW VARIANTS

WORKFLOW A Explore, then build

the common case

Query → Brief → B → merge

WORKFLOW B Significant feature

new model · integration · boundary

Brief 1 · spec → Brief 2 · build

WORKFLOW C Narrow / parallel

operator says “too big”

split → B1…Bn · worktrees → Brief M · merge

WORKFLOW D Hands-off / headless

Approve & queue (auto)

queue-cli add → daemon → Q / D manage

SECTION 09

Prompt engineering, not chatting

The same request, run two ways — ad-hoc prose versus a governed, versioned pipeline.

DIAGRAM 12 · BEFORE / AFTER

VIBE PROMPTING	REPO OS ENGINEERED PROMPTS
Prompt = prose typed each time	Prompt = pointer to a versioned, governed file
Rules live in the operator's head	Rules live in AI.md + hooks + decisions
Plan quality = one model's guess	Plan quality = advisory board + synthesis
Approval = "looks good" in chat	Approval = AskUserQuestion → state file → unblocks edits
Handoff = re-explain to the next session	Handoff = sealed bundle + drift_check forward-jump
Enforcement = hope	Enforcement = mechanical hooks that block commits

THE THESIS

Repo OS treats prompts as engineered, versioned artifacts that flow through a governed pipeline — spec → advisory review → approved brief → gated build → merge — where an AI specialist board does the planning and a hook layer makes the discipline non-optional. The operator's job shrinks to direction and approval; the system's job is structure, review, and enforcement.

APPENDIX

Key file references

For accuracy when reading the diagrams — written from inside the repo, so paths and Decision numbers are exact.

> **SOURCE OF TRUTH**
> AI.md · docs/ai/prompts/* · .claude/agents/orchestrator.md · .claude/agents/_advisory-board-lite.md · docs/ai/workflow-summary.md

Prompt library	docs/ai/prompts/prompt-{0,0.5,a,b,c,d,e,f,h,q,t}-*.md
Governance index	AI.md
Orchestrator contract	.claude/agents/orchestrator.md
Advisory board format	.claude/agents/_advisory-board-lite.md (full: _advisory-board.md)
Phase model / helpers	.claude/hooks/_phase-helper.sh · _agent-glossary.md §Phase Model
Handoff bundle	.claude/local/handoff-bundles/<brief_id>.yaml (Decision 0038, t-0043)
Hook chain	AI.md §Hooks table
Compiled specs	docs/spec/compiled/{active}/ (resolved via manifest.json)
Key decisions	0014 phase model · 0035 selective spec loading · 0038 handoff / drift-check · 0033 worktree isolation · 0025 Pragmatist plan-gating · t-0043 per-brief bundles