real world technology solutions

Version 1.1 (Drafting)

Implement Skill — Specification

Implement Skill

Project	Implement Skill
Entity	Real World Technology Solutions Pty Ltd
Purpose	Specification Document

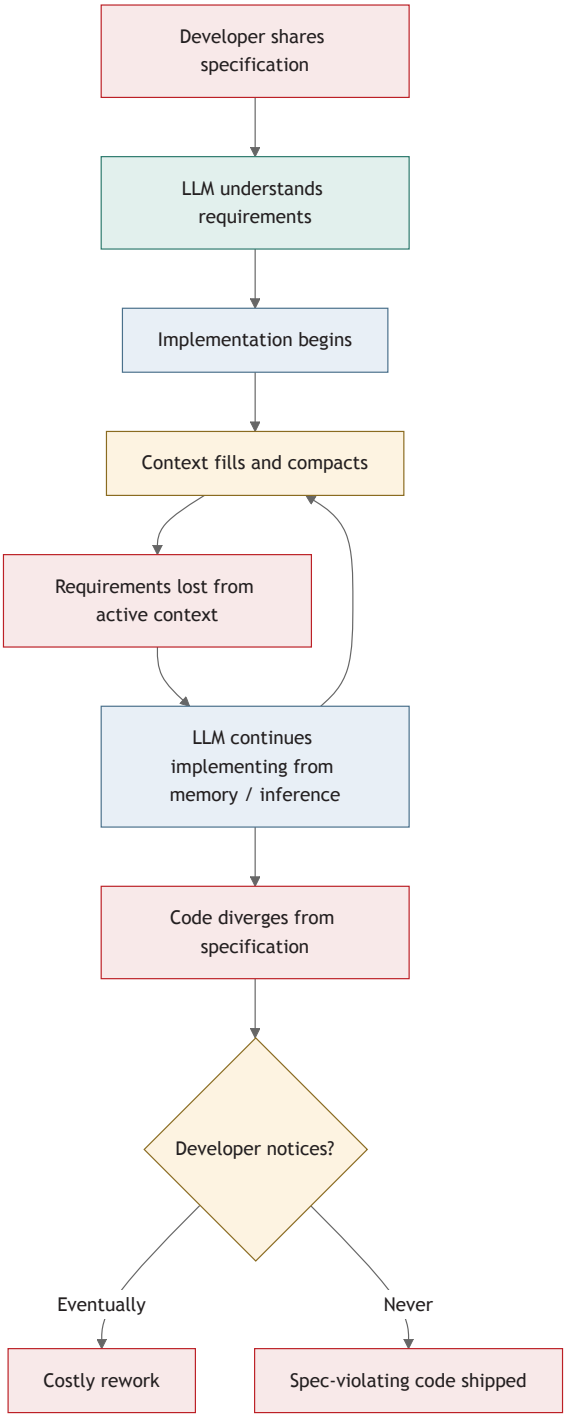
Contents

§1 Overview

§1.1 Problem Statement

Modern LLM-assisted development involves extended, multi-turn implementation sessions driven by specification documents. In practice, a systematic failure pattern emerges: the LLM begins with a clear understanding of the specification, but as implementation progresses and the conversation context fills and compacts, specific requirements and constraints are silently lost. The LLM continues producing code — confidently, fluently — while drifting away from the specification. Neither the developer nor the LLM is necessarily aware this is happening.

This failure mode is called **specification drift**.



Specification drift is not a failure of intent — it is a structural consequence of how LLMs process long-context inputs. Research demonstrates that:

- LLMs exhibit a **U-shaped attention curve**: performance is strongest at the beginning and end of the context window and degrades significantly in the middle.
- Performance on retrieval-dependent tasks degrades **13–85%** as input length increases, even when the required information is technically present.

- Multi-turn conversations show an average **39% performance drop** compared to single-turn equivalents, as earlier turns are compressed or evicted.

These effects compound over an implementation session. By the time 50 or 100 requirements have been worked through, the LLM's connection to requirements discussed early in the session may be negligible.

---

§1.2 Goals

The `/implement` skill is designed to make specification drift structurally impossible, rather than relying on the LLM's in-context attention to maintain fidelity.

The skill aims to:

1. **Maintain a persistent connection** between specification documents and implementation work that survives context compaction events, session restarts, and model switches.
2. **Prevent specification drift** through structured requirement tracking, section reference anchoring, and mandatory spec re-reads at the start of each implementation unit.
3. **Enable systematic verification** of every individual requirement against the implementation, with clear pass/fail reporting and delta tracking between verification runs.
4. **Support both standard and TDD workflows**, exploiting the context isolation between sub-agents to cleanly separate test authoring from implementation.
5. **Scale to large projects** — the methodology must remain effective for specifications with hundreds or thousands of requirements, where manual tracking is not feasible.

---

§1.3 Scope

In Scope

- **Five-phase workflow**: Planning → Implementation → Verification → Status → Continue
- **Persistent tracker files** with embedded self-recovery instructions so the orchestrator can reconstruct context after compaction
- **Sub-agent orchestration** with model tiering (haiku for routine tasks, sonnet for implementation, opus for complex reasoning)
- **TDD mode** that exploits context isolation between agents to write tests before implementation
- **Per-requirement verification** with V-item tracking and delta reporting between runs
- **Git worktree support** for concurrent, isolated implementations of multiple features
- **Preferences system** with both project-scope and global-scope configuration
- **Commands**: `/implement`, `/implement verify`, `/implement continue`, `/implement status`, `/implement list`, `/implement config`

Out of Scope

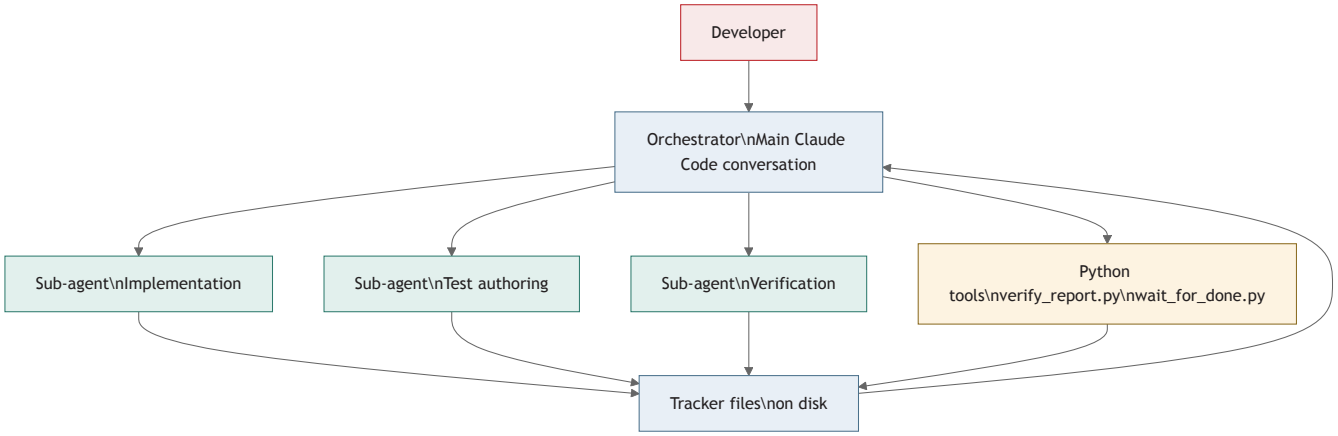
Area	Rationale
Writing specifications	That is the responsibility of the `/spec` skill
Code deployment and CI/CD integration	Out of the implementation assistant's domain
Project management and issue tracker integration	Spec-to-code is the boundary; issue lifecycle is separate

Area	Rationale
Comprehensive code quality review	Limited quality checks exist, but spec verification is the primary purpose
Refactoring workflows	Possible in principle, but not the skill's strength or design target

---

§1.4 Key Actors

The skill involves four distinct actors that interact during an implementation session.



Developer The human user who invokes the skill, provides or points to a specification document, and reviews implementation output. The developer is the final arbiter of correctness and makes decisions when the skill surfaces ambiguities or failures.

Orchestrator The main Claude Code conversation that manages the overall workflow. The orchestrator reads tracker files, delegates tasks to sub-agents, interprets results, and presents status summaries to the developer. It does not perform implementation work directly; it coordinates.

Sub-agents Fresh Claude instances spawned with isolated context windows. Each sub-agent receives a focused task brief that includes the relevant specification sections and implementation context. Context isolation is the key mechanism: because sub-agents start with a clean context, they cannot have drifted from the specification during prior turns. Sub-agents write their results as structured output to disk rather than returning conversational responses.

Python tools Deterministic scripts (`verify_report.py`, `wait_for_done.py`) that perform assembly, polling, and coordination tasks that are better handled by code than by LLM inference. These tools process sub-agent output files and produce structured verification reports.

---

§1.5 Design Origin

The `/implement` skill did not emerge from theory — it was built through direct observation of failure modes in LLM-assisted development.

The journey began with a straightforward approach: share a specification document with Claude, then ask it to implement the features. This worked well initially. For small specifications with a handful of requirements, the LLM maintained fidelity. But as projects grew — specifications running to dozens of sections, implementations spanning multiple sessions — a consistent pattern of silent failure emerged. Features would be implemented that seemed correct but violated specific constraints documented in the middle of a long spec. The LLM would not flag this. It would not ask for clarification. It would proceed with confidence.

Over the course of approximately ten documented iterations, the methodology evolved through a series of failure modes and their corresponding countermeasures:

- **Iteration 1–3**: Manual specification-driven implementation. Identified the core drift problem.
- **Iteration 4–5**: Mandatory test execution and hard enforcement gates added. Testing served as a synchronisation point catching both mechanical errors and spec misunderstandings. The discovery that LLMs skip optional steps under context pressure led to replacing guidelines with mandatory preconditions.
- **Iteration 6–7**: Context-aware activation and dedicated test-writing with quality controls. Separating test writing from implementation into its own sub-agent — working from the spec alone, never seeing implementation code — produced a significant quality improvement.
- **Iteration 8**: Per-requirement verification granularity. Counter-intuitively, spawning one sub-agent per requirement (20–40+ agents) proved both more thorough and faster than batching multiple requirements per agent.
- **Iteration 9**: TDD mode added, exploiting context isolation between agents — the test-writing agent and implementation agent independently interpret the specification, surfacing ambiguities.
- **Iteration 10**: Large specification handling with structural indexing, size-based model routing, and the full five-phase workflow formalised.

Each iteration was driven by a specific, observed failure. The design reflects accumulated empirical evidence about where LLM-assisted implementation breaks down and what structural interventions actually prevent it.

The result is a skill built around one core principle: **do not rely on the LLM's in-context attention to maintain specification fidelity**. Make the connection between specification and implementation structural, persistent, and verifiable.

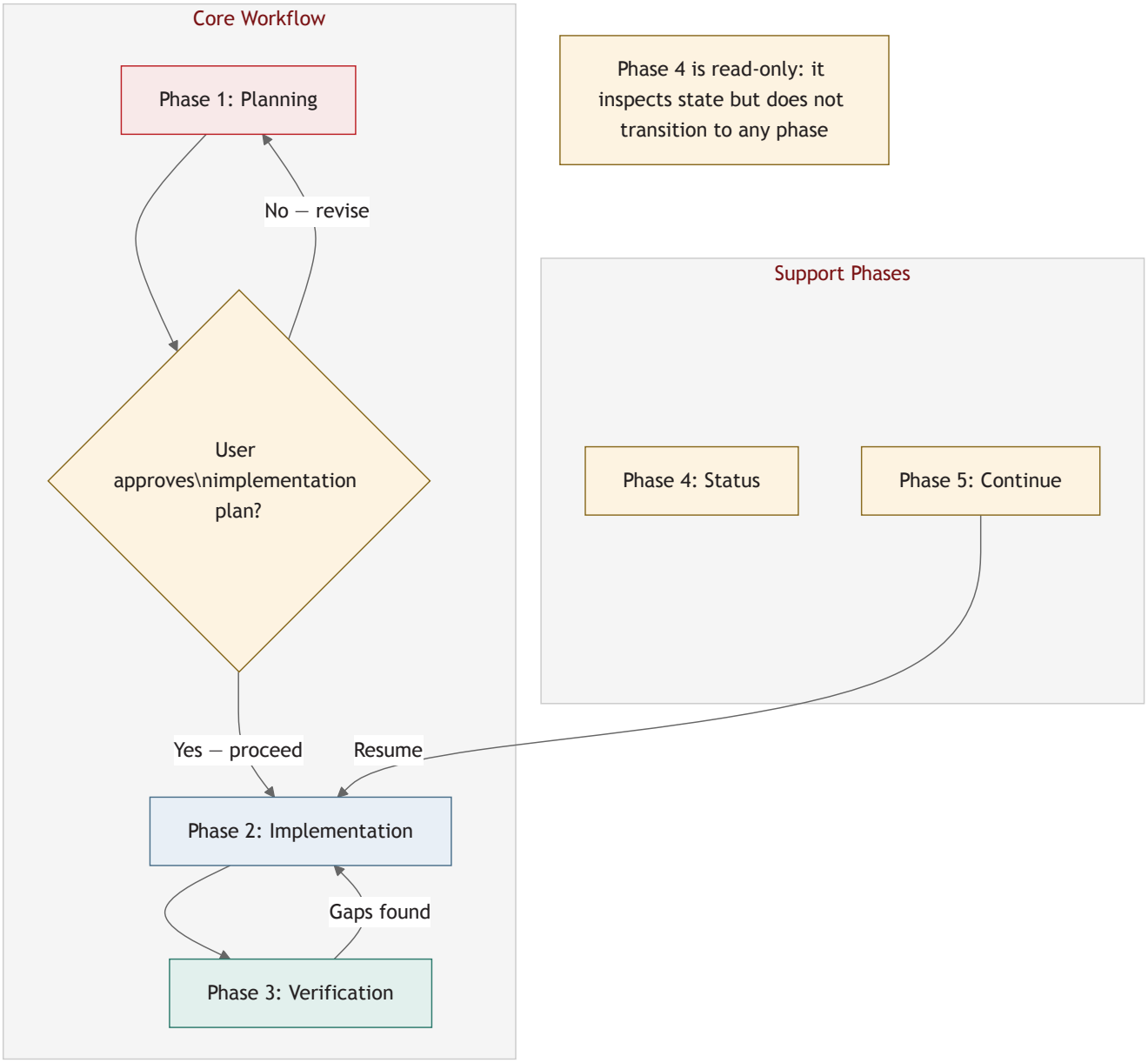
§2 Architecture & Design Principles

This section documents the architectural decisions and design principles behind the `/implement` skill. Every principle described here originated from observed failure modes during iterative development — they are empirical findings, not theoretical preferences. This is the most critical section of the specification: it captures the **why** behind the skill's design, without which the functional requirements in later sections would appear arbitrary.

---

§2.1 Architectural Overview

The skill operates as a five-phase workflow, orchestrated by the main Claude Code conversation (the orchestrator) and executed through delegated sub-agents (isolated Claude Code instances spawned via the Task tool, each with a fresh context window — see §2.2, Principle 3 and §2.3) and deterministic tooling.



Phase	Command	Purpose
Phase 1: Planning	`/implement <spec-path>`	Parse spec, create tracker, break into tasks, determine TDD mode (test-driven development — see §2.2, Principle 3)
Phase 2: Implementation	*(begins after user approves the implementation plan)*	Standard or TDD workflow via sub-agent delegation
Phase 3: Verification	`/implement verify`	Per-requirement verification with parallel sub-agents, deterministic report assembly
Phase 4: Status	`/implement status`	Read-only view of progress, blockers, and current state

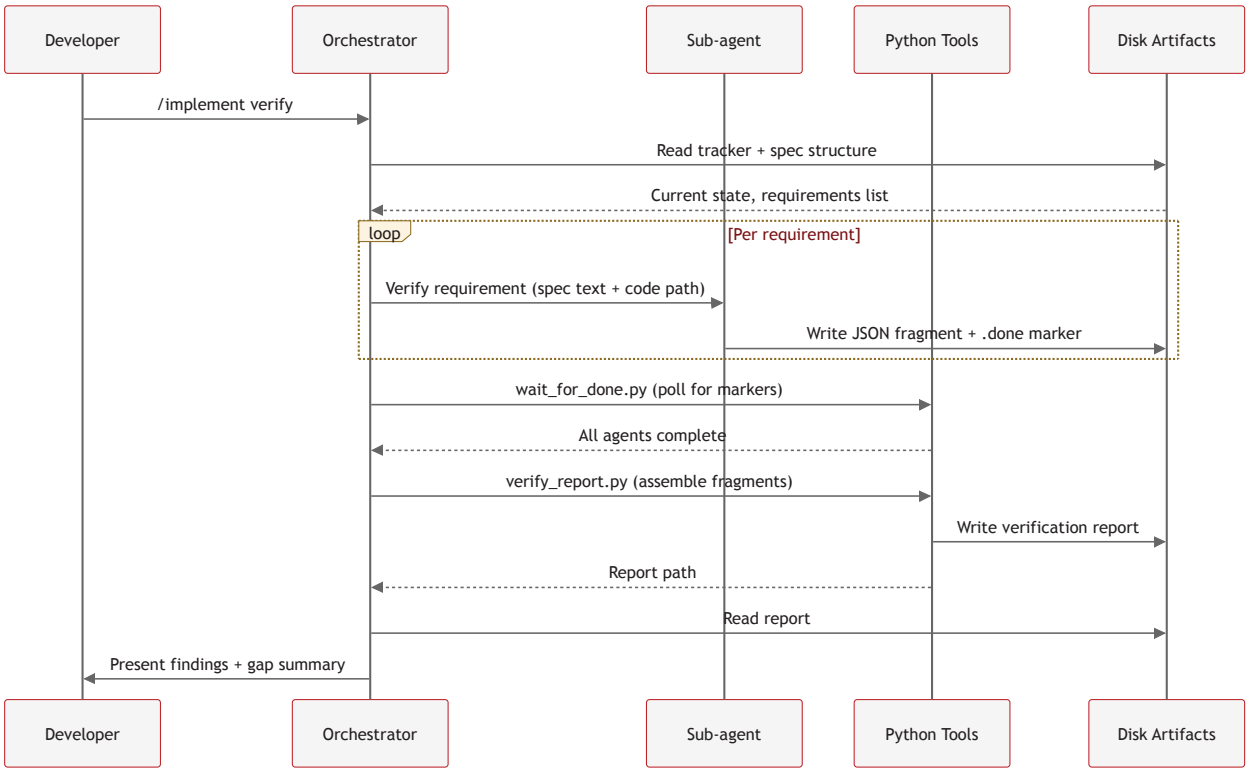
Phase	Command	Purpose
Phase 5: Continue	`/implement continue`	Resume from previous session with worktree validation and spec freshness checks

Phases 1 through 3 form the core workflow: plan, implement, verify. Phase 4 (Status) is a read-only inspection that can be invoked at any point. Phase 5 (Continue) is the session-recovery entry point that feeds back into Phase 2, re-establishing context from the persistent tracker before resuming implementation.

The verification phase loops back to implementation when gaps are found. This cycle — implement, verify, fix gaps, re-verify — continues until all requirements are satisfied or explicitly documented as out of scope.

§2.1.1 Data Flow Through Phases

Within each phase, data flows through a consistent pattern involving three actors: the orchestrator, sub-agents, and deterministic tools.



This pattern — orchestrator prepares context, sub-agents produce structured fragments, tools assemble deterministically, orchestrator reasons over the result — recurs throughout the skill and is formalised in §2.4 as the Model-Tool-Model Pipeline.

---

§2.2 Core Design Principles

The following eight principles govern the skill's architecture. Each originated from a specific, observed failure during iterative development (see §1.5 for the iteration history). They are presented in order of architectural significance.

Principle 1: Context Window as the Fundamental Constraint

Empirical origin: Every failure mode observed during the skill's development traced back, directly or indirectly, to context window limitations — information loss during compaction, attention degradation over long inputs, or context exhaustion during complex tasks.

Statement: The context window MUST be treated as the fundamental design constraint. Every architectural decision — file structure, delegation strategy, artifact format, workflow sequencing — MUST be evaluated against its impact on context consumption and resilience to context loss.

Implications:

- The skill's own SKILL.md was restructured from a monolithic document (approximately 1400 lines at the time of restructure) to a slim routing document (approximately 296 lines at the time of restructure — actual current line count may vary; see §2.5) — a context optimisation applied to the skill itself.
- Reference files are loaded on demand per phase, never all at once.
- Sub-agents receive only the spec sections and code paths relevant to their specific task, not the full specification.
- Tracker files are designed for efficient re-reading: structured headers, consistent formatting, and embedded recovery instructions allow rapid context reconstruction.
- Verification dispatches one requirement per sub-agent rather than batching, because smaller focused contexts produce more thorough analysis than larger overloaded ones.

Principle 2: Hard Enforcement Gates Over Soft Guidelines

Empirical origin: During iterations 4-5, it was observed that LLMs consistently skip optional steps when under context pressure. Instructions phrased as "consider doing X" or "you may want to check Y" are treated as suggestions — and suggestions are the first things dropped when context fills. This behaviour is consistent: across models, across prompt styles, across task types.

Statement: Requirements MUST be expressed as mandatory preconditions, not suggestions. The skill MUST use hard gates — conditions that block progress until satisfied — rather than soft guidelines that can be bypassed.

Examples of hard gates in the skill:

Gate	Blocks	Rationale
Tracker must exist before implementation begins	Phase 2	Without a tracker, there is no recovery mechanism after compaction
Tests must pass before marking a task complete	Task completion	"Complete but tests failing" is a contradiction the LLM will not self-correct
Tests must pass before verification begins	Phase 3	Verifying non-functional code produces meaningless results
Opus must be used for verification fixes	Gap fixing	Sonnet-class models miss subtle spec gaps that Opus catches

Gate	Blocks	Rationale
DIGEST escalation (see §5.4 for details) to Opus is mandatory, not discretionary	Post-task review	"Consider reviewing" means "don't review" under context pressure

Analogy: Hard gates are the equivalent of compile-time errors; soft guidelines are runtime warnings. In safety-critical contexts, compile-time enforcement is always preferred.

Principle 3: Context Isolation as a Verification Advantage

Empirical origin: During iteration 6-7, separating test writing into its own sub-agent — one that reads only the specification and never sees implementation code — produced a measurable quality improvement. Tests written by an agent that has seen the implementation tend to test what the code does rather than what the spec requires. The "fresh eyes" of an isolated context revealed ambiguities and gaps that were invisible to the implementing agent.

Statement: Sub-agent context isolation MUST be treated as a feature, not a limitation. Fresh context windows provide independent specification interpretation that is structurally incapable of implementation drift.

Applications:

- **TDD mode**: The test-writing agent reads only the spec. The implementing agent receives the test file path but writes code to satisfy the tests. The two agents independently interpret the specification, and disagreements surface as test failures — which are ambiguities that need human resolution.
- **Verification**: Each verification sub-agent starts fresh, reads the spec requirement and the implementation code, and produces an independent judgment. It cannot have been influenced by the implementing agent's reasoning or assumptions.
- **Fix agents**: When fixing verification gaps, the fix agent receives the spec quote, the current code, and the gap description. It does not inherit the original implementer's context or biases.

Principle 4: Three Layers of Context Preservation

Empirical origin: No single recovery mechanism proved sufficient on its own. Tracker files could become stale. Sub-agent delegation could fail. Section references could be ambiguous. But the combination of all three provided resilient recovery — any single layer failing still allowed reconstruction via the others.

Statement: The skill MUST maintain three independent layers of context preservation. Any single layer failing MUST still allow recovery through the remaining two.

This principle is significant enough to warrant its own subsection — see §2.3 for the full treatment.

Principle 5: The Model-Tool-Model Pipeline

Empirical origin: Early iterations had LLMs both generate verification findings and assemble them into reports. The assembly step — collecting fragments, computing statistics, formatting output — is a deterministic task that LLMs perform unreliably. They hallucinate counts, misformat tables, and silently drop items. Moving assembly to a Python script eliminated an entire class of errors.

Statement: The skill MUST separate LLM judgment from deterministic assembly. Models SHOULD produce structured data (JSON fragments). Deterministic tools MUST assemble that data. Models SHOULD then reason over the assembled results.

See §2.4 for the full treatment of this pattern.

Principle 6: Self-Recovering Artifacts

Empirical origin: After total context loss (session restart, severe compaction), the orchestrator needs to reconstruct its understanding of the implementation state. If recovery depends on external documentation or human explanation, the skill fails its core purpose. The tracker must be self-explanatory.

Statement: The implementation tracker MUST embed its own recovery instructions. After total context loss, reading the tracker alone MUST provide sufficient information to understand the current state, determine the next action, and resume work without external assistance.

Implementation: Every tracker file contains a Recovery Instructions section at the top that tells the reader:

- What this file is and what it tracks
- Where the specification is located
- What phase the implementation is in
- How to read the requirements matrix
- What to do next

This creates zero dependency on external infrastructure for recovery. The tracker is a self-contained resumption artifact.

Principle 7: Consent-Based Activation

Empirical origin: Early versions of the skill activated automatically when it detected spec-like documents or tracker files in the conversation. This frustrated users who were performing unrelated tasks near spec files. The skill was "helpful" in a way that disrupted the user's actual intent.

Statement: The skill MUST detect opportunities for activation but MUST ask before engaging. Implicit activation (detecting tracker files, spec references, or implementation-related language) MUST prompt the user for confirmation before proceeding.

The rule: The skill offers help; it does not hijack the conversation. User agency is preserved at every activation point.

Principle 8: Section References as Stable Anchors

Empirical origin: During context compaction, prose descriptions of requirements are among the first things lost. A phrase like "the requirement about handling concurrent barcode scans in the quick capture module" is verbose, ambiguous, and fragile under compression. The compact reference "§2.1.3" survives compaction reliably and can be resolved by re-reading the spec.

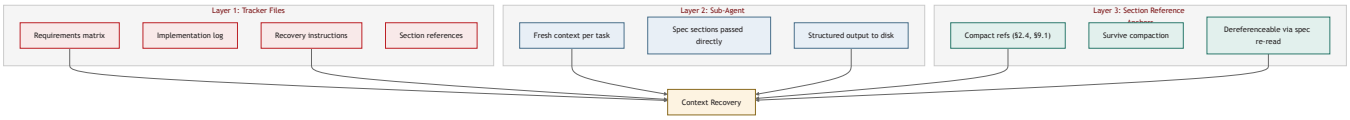
Statement: Every task, tracker entry, and verification item MUST reference specific spec sections using compact section references (e.g., §2.4, §9.1). These references SHOULD survive context compaction where prose descriptions do not.

Why this works: Section references are short (4-6 characters), unambiguous, and semantically dense. They serve as pointers that can be dereferenced by reading the spec — the information itself does not need to survive in context, only the pointer to it. This is analogous to how memory addresses work: the address is small and stable; the data it points to can be arbitrarily large.

---

§2.3 Three-Layer Context Preservation

Context preservation is the central challenge the skill addresses. Three independent mechanisms work together to ensure that specification fidelity survives context compaction, session restarts, and model switches.



Layer 1: Persistent Tracker Files

The tracker file (`.impl-tracker-<name>.md`) is the primary persistence mechanism. It is a structured markdown file stored on disk that survives any context event — compaction, session restart, model switch, or complete conversation loss.

The tracker contains:

- **Metadata**: spec path, worktree path, TDD mode, creation date
- **Recovery instructions**: self-contained guidance for resuming after context loss
- **Requirements matrix**: every requirement with status, section reference, and implementation location (file:line)
- **Implementation log**: chronological record of work performed
- **Structural index**: for multi-file specs, a size-based index of section files

Because the tracker is on disk, it is immune to context window events. The cost is that it must be explicitly read — it does not "persist" in the LLM's attention automatically. This is why the pre-task checklist mandates reading the tracker before each implementation unit.

Layer 2: Sub-Agent Delegation

Each sub-agent receives a fresh context window containing only the information relevant to its specific task: the spec sections being implemented, the target code paths, and the task brief. This fresh context is structurally incapable of drift — the sub-agent has no prior conversation history to have drifted from.

Sub-agents write their results as structured JSON to disk rather than returning conversational responses. This means the results persist independently of the sub-agent's context window and can be read by the orchestrator or by deterministic tools at any point.

The combination of fresh-context input and structured-output-to-disk means that sub-agent delegation is itself a context preservation mechanism: it converts ephemeral conversation context into persistent disk artifacts.

Layer 3: Section Reference Anchors

Section references (§2.4, §9.1, Section 3.2) are compact identifiers that survive context compaction where prose descriptions do not. They appear in:

- Task descriptions and subjects
- Tracker requirements matrix entries
- Verification item identifiers
- Sub-agent task briefs
- Implementation log entries

After compaction, even if the orchestrator has lost the detailed understanding of what §4.2 requires, the reference itself survives. The orchestrator can dereference it by reading that section of the spec — reconstructing full understanding from a minimal surviving anchor.

Recovery Scenarios

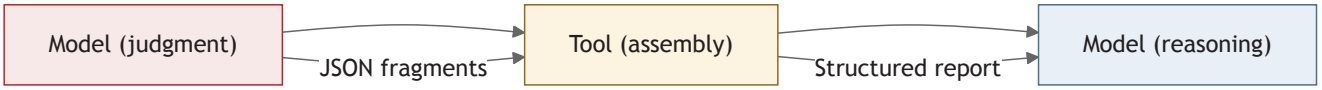
Scenario	Layer 1 (Tracker)	Layer 2 (Sub-agents)	Layer 3 (References)	Recovery Path
Mid-session compaction	Available on disk	Current agent unaffected	Survive in tracker	Read tracker, continue
Session restar	Available on disk	N/A (new session)	Survive in tracker	Phase 5: Continue
Tracker accidentally deleted	Lost	Output files on disk	In task descriptions	Reconstruct from `.impl-work/` artifacts and task list
Context completely exhausted	Available on disk	Spawn fresh agent	Survive in tracker	Read tracker recovery instructions
Spec file relocated or renamed	Available on disk (old path recorded)	N/A	Section refs intact but path stale	Update spec path field in tracker header; verify section refs still resolve; use Phase 5 Continue to re-establish context

The design intent is that no single point of failure can cause unrecoverable context loss. The three layers provide defence in depth.

---

§2.4 The Model-Tool-Model Pipeline

The Model-Tool-Model pipeline is the skill's core execution pattern. It separates what LLMs are good at (judgment, analysis, interpretation) from what deterministic tools are good at (assembly, counting, formatting).



The Pattern

1. **Model produces structured data**: A sub-agent (LLM) reads the specification and implementation code, exercises judgment about whether a requirement is satisfied, and writes its finding as a JSON fragment to disk. The model does what models are good at: interpreting natural language requirements, reading code, and making qualitative judgments.
2. **Tool assembles deterministically**: A Python script (`verify_report.py`) collects all JSON fragments, validates their structure, computes aggregate statistics, identifies priority gaps, and produces both machine-readable

(JSON) and human-readable (Markdown) reports. The tool does what tools are good at: deterministic data processing with no hallucination, no miscounting, and no silent omissions.

3. **Model reasons over assembled results**: The orchestrator reads the assembled report and presents findings to the developer, identifies patterns across gaps, and plans remediation. The model does what models are good at: synthesis, prioritisation, and communication.

Why Not Let the Model Do Everything?

Early iterations attempted to have LLMs perform the assembly step. The failure modes were consistent:

- **Miscounting**: An LLM summarising 40 verification results would report "38 of 40 passed" when the actual count was 35 of 40.
- **Silent omission**: Requirements whose findings were ambiguous or complex would be quietly dropped from the summary.
- **Format drift**: Report formatting degraded as context filled, with tables losing columns or entries being merged.
- **Non-reproducibility**: The same inputs could produce different assembly results across runs.

These are not edge cases — they are the expected behaviour when asking a probabilistic system to perform a deterministic task. The pipeline eliminates this class of errors by reserving deterministic work for deterministic tools.

Where the Pipeline Appears

Phase	Model Step (judgment)	Tool Step (assembly)	Model Step (reasoning)
Verification	Sub-agents write per-requirement JSON fragments	`verify_report.py` assembles report	Orchestrator interprets report, plans fixes
Sub-agent coordination	Orchestrator dispatches tasks and constructs prompts	`wait_for_done.py` polls `.done` markers	Orchestrator proceeds when all complete
Re-verification	Sub-agents write updated JSON fragments	`verify_report.py` with `--previous` flag computes deltas	Orchestrator presents delta summary
Planning (Phase 1)	Orchestrator parses spec sections and structures tasks	`parse_spec.py` extracts requirements and builds tracker	Orchestrator reviews tracker before seeking user approval

---

§2.5 Skill Structure: Routing Document and Reference Files

The skill's own file structure embodies the context management principles it teaches. In February 2026, the SKILL.md file was restructured from a monolithic document (approximately 1400 lines at that time) to a slim routing document (approximately 296 lines at that time; actual current counts may differ) that loads phase-specific reference files on demand.

The Problem with Monolithic Skill Files

When SKILL.md contained the full workflow documentation — every phase's detailed steps, every prompt template reference, every edge case — it consumed a substantial portion of the context window at activation time. This meant that by the time the skill had loaded its own instructions, it had already reduced the available context for actual specification content and implementation work.

This is the same problem the skill solves for specifications: a large document loaded entirely into context leads to information loss through compaction pressure.

The Routing Document Pattern

The restructured SKILL.md serves as a **routing document**: it contains just enough information to understand the skill's purpose, determine which phase to enter, and know which reference file to load for detailed instructions.

```
# Path shown relative to the Claude Code skills root (e.g. ~/.claude/skills/ or the project's .claude/skills/)
skills/implement/
  SKILL.md                          # ~296 lines — routing document
  references/
    tracker-format.md               # Full tracker template + field explanations
    workflow-quick-ref.md           # Checklists for pre-task, post-task, verification
    sub-agent-strategy.md           # Model selection, size routing, DIGEST escalation
    phase-1-planning.md             # Full Phase 1 workflow
    phase-2-implementation.md       # Standard implementation workflow
    phase-2-tdd.md                  # TDD test-first workflow
    phase-3-verification.md         # Full verification machinery
    phase-5-continue.md             # Resume work + spec evolution handling
  prompts/
    implement-single-file.md        # Sub-agent prompt templates
    implement-multi-file.md
    write-tests.md
    fix-issue.md
    tdd-write-tests.md
    tdd-implement.md
    verify-requirement.md
    reverify-requirement.md
    fix-verification-gap.md
    # Note: this list reflects the prompts directory as of the February 2026 restructure.
    # The actual directory is authoritative — new prompt files may have been added since.
```

The routing document contains:

- Command table and argument routing logic
- Preference system documentation
- Phase summaries with "read this reference file" pointers
- Recovery instructions
- Implicit activation rules
- Best practices (concise)

The routing document does not contain:

- Detailed step-by-step workflows (those are in `references/phase-*.md`)
- Prompt templates (those are in `prompts/`)

- Tracker format specification (that is in `references/tracker-format.md`)
- Model selection tables and escalation logic (that is in `references/sub-agent-strategy.md`)

Load-on-Demand Principle

The instruction in SKILL.md is explicit: **"Load only what you need for the current phase. Do not read all files at once."**

When the orchestrator enters Phase 1, it reads `references/phase-1-planning.md`. When it enters Phase 3, it reads `references/phase-3-verification.md`. Reference files for other phases remain unloaded, preserving context for the work at hand.

This is the same principle as the tracker's section references: don't load information into context until it is needed. The reference file path serves as a pointer — small, stable, and dereferenceable on demand — just as a section reference like §4.2 serves as a pointer to a spec requirement.

The Skill Eating Its Own Dogfood

The restructure of SKILL.md is a direct application of the skill's own design principles to itself:

Principle	Application in Skill Structure
Context window as fundamental constraint (§2.2, Principle 1)	Reduced from ~1400 lines (monolithic) to ~296 lines (routing document) at the time of restructure — substantially smaller activation footprint
Section references as stable anchors (§2.2, Principle 8)	Phase summaries point to reference files by path
Hard enforcement ("load only what you need") (§2.2, Principle 2)	Explicit instruction, not a suggestion
Self-recovering artifacts (§2.2, Principle 6)	Routing document contains enough to determine which reference to load

This self-application demonstrates that the design principles are general — they apply to any situation where an LLM must work with more information than fits comfortably in a single context window.

§3 Functional Requirements

This section specifies the functional requirements for each phase, command, and activation mode of the `/implement` skill. Requirements use MoSCoW language (MUST, SHOULD, COULD, WON'T) throughout and are numbered for traceability during verification.

> **Requirement status values**: Throughout this section, requirement statuses use the canonical set defined in `references/tracker-format.md`: `pending`, `in_progress`, `partial`, `complete`, `blocked`, `n/a`. All requirements start as `pending` during Phase 1 planning. No other status values are valid.

---

§3.0 Common Initialization

Common Initialization runs before every phase — `/implement <spec-path>`, `/implement continue`, `/implement verify`, `/implement status`, `/implement list`, `/implement config`. It establishes the runtime environment that all subsequent phases depend on.

§3.0.1 Tool Path Resolution

- **FR-0.1** The skill MUST resolve three tool path variables during Common Initialization: `IMPL_REPO_DIR`, `IMPL_TOOLS_DIR`, and `IMPL_PYTHON`. These MUST be resolved by following the SKILL.md symlink to the skill repository. See §6.3 Tool Discovery at Runtime for the canonical resolution commands and platform notes.
- **FR-0.2** `IMPL_PYTHON` MUST point to the skill repository's virtual environment Python (`$IMPL_REPO_DIR/.venv/bin/python`), not the system `python3`. This ensures consistent Python version and dependency availability.
- **FR-0.3** The skill MUST verify that `$IMPL_PYTHON` exists on disk after resolution. If it does not exist, the skill MUST warn the user that the skill repository's venv is missing and suggest running `python3 -m venv .venv` in the skill repo.
- **FR-0.4** All subsequent bash commands in all phases MUST use `$IMPL_REPO_DIR`, `$IMPL_TOOLS_DIR`, and `$IMPL_PYTHON`. No phase may redefine these variables with different names (e.g., bare `$PYTHON`, `$TOOLS_DIR`, `$REPO_DIR`).

§3.0.2 Sub-Agent Write Permissions

Sub-agents dispatched with `run_in_background: true` cannot prompt the user for file write permissions. Without pre-configured scoped permissions, background sub-agents fail silently when attempting to write output files (`.impl-work/`, `.impl-verification/`, `.impl-tracker-*.md`).

- **FR-0.5** The skill MUST check `.claude/settings.local.json` for scoped write permissions before any phase that dispatches file-writing sub-agents.
- **FR-0.6** The required scoped permissions MUST be:
  - `Edit(/.impl-work/**)`
  - `Write(/.impl-work/**)`
  - `Edit(/.impl-verification/**)`
  - `Write(/.impl-verification/**)`
  - `Edit(/.impl-tracker-*.md)`
  - `Write(/.impl-tracker-*.md)`
- **FR-0.7** If all required permissions are present, the skill MUST skip permission setup (no-op).
- **FR-0.8** If any permissions are missing, the skill MUST inform the user what it will add and why, then write/merge the missing entries into `settings.local.json`, preserving all existing entries. The expected JSON structure is:

```
{
  "permissions": {
    "allow": [
      "Edit(/.impl-work/**)",
      "Write(/.impl-work/**)",
      "Edit(/.impl-verification/**)",
      "Write(/.impl-verification/**)",
      "Edit(/.impl-tracker-*.md)",
      "Write(/.impl-tracker-*.md)"
    ]
  }
}
```

The merge operation MUST preserve any existing entries in the `permissions.allow` array and append only the missing entries. If `settings.local.json` does not exist, create it with only the `permissions` block above.

- **FR-0.9** The skill MUST NOT add unscoped `Write` or `Edit` permissions. Permissions MUST be limited to implementation output paths only. **Note on Bash tool writes**: Claude Code's `Write` and `Edit` scoped permissions cover only the Write and Edit tools. Sub-agents that create directories via the Bash tool (e.g., `mkdir -p .impl-work/`) are not restricted by these scoped permissions — Bash tool permissions are governed separately. Directory creation via Bash within the scoped paths is expected and does not require additional permission entries.
- **FR-0.10** If the user denies the settings write, the skill MUST log a warning and proceed. Sub-agents may fail silently in this case, but the user has made an informed choice.

§3.0.3 Worktree-Aware Permission Setup

Git worktrees (created via `git worktree add`) have their own project root but do NOT inherit the main tree's `.claude/settings.local.json`. Claude Code resolves settings relative to `git rev-parse --show-toplevel`, which returns the worktree root, not the main tree root.

> **Disambiguation**: This worktree detection determines the *current runtime environment* for permission file placement. It is distinct from §3.1.2's worktree detection, which determines the spec's intended *implementation directory*. §3.0.3 asks "where am I running?" while §3.1.2 asks "where does the spec want me to implement?"

- **FR-0.11** During Common Initialization, the skill MUST detect whether the current working directory is inside a git worktree by comparing `git rev-parse --show-toplevel` with the main tree root derived from `git rev-parse --git-common-dir`. If `git rev-parse` fails (git unavailable or not a git repository), the skill MUST skip worktree detection, treat the current working directory as the project root for permission setup purposes, and log a warning.
- **FR-0.12** If in a worktree, the skill MUST ensure `.claude/settings.local.json` exists in the **worktree root**, not the main tree root. The `.claude/` directory MUST be created if it does not exist (`mkdir -p .claude`).
- **FR-0.13** If the main tree already has a `settings.local.json`, the skill SHOULD use it as a starting point (read it, then merge in the required permissions from FR-0.6).
- **FR-0.14** If not in a worktree, the skill MUST read and update `.claude/settings.local.json` relative to the project root as normal.
- **FR-0.15** If any Common Initialization step fails for a reason not covered by a specific FR (e.g., broken SKILL.md symlink, malformed `settings.local.json`, unexpected `git rev-parse` output), the skill MUST warn the user with the specific error, suggest a remediation action, and MUST NOT proceed to phase execution.

---

§3.1 Phase 1: Planning (`/implement <spec-path>`)

Phase 1 transforms a specification document into an actionable implementation plan: a persistent tracker file, a task breakdown, and a determined workflow mode. No implementation code is written during this phase.

> **Note on tracker-level status**: Tracker lifecycle status transitions (Planning → In Progress → Verification → Complete) are documented in §4.7.1 as an advisory lifecycle. §3 does not define FRs for tracker-level status because transitions are managed by the orchestrator implicitly, not enforced programmatically.

§3.1.1 Context Clear Offer

- **FR-1.1** The skill MUST offer to clear conversation context before beginning planning work. The offer MUST explain the benefit (maximum context window for implementation).
- **FR-1.2** If the user declines, the skill MUST proceed without asking again.
- **FR-1.3** If the skill is re-invoked after a `/clear`, the skill MUST NOT repeat the context clear offer. Note: the skill intentionally loses all in-memory state after `/clear` — there is no persistent detection mechanism. This FR describes the behaviour when the skill detects an existing tracker on re-invocation (per Phase 5 / Continue flow), not a `/clear`-specific detection mechanism.

§3.1.2 Worktree Detection and Validation

- **FR-1.4** After reading the specification document, the skill MUST scan for any indication of a worktree path, working directory, or branch designation for implementation work. This scan MUST be generic — not tied to specific field names or document formats. Minimum detection criteria: scan for mentions of absolute or relative directory paths, git branch names, or worktree-related keywords (e.g., "worktree", "working directory", "implementation directory") in the spec/brief document. This is intentionally heuristic, not a formal parser.
- **FR-1.5** If a worktree path is identified, the skill MUST validate that:
  1. The path exists on disk.
  2. The path appears in `git worktree list` output.
  3. If a branch was specified, the worktree is on that branch (verified via `git -C <path> branch --show-current`).
- **FR-1.6** If validation fails, the skill MUST warn the user and ask whether to proceed in the current directory or abort. The skill MUST NOT silently fall back.
- **FR-1.7** If a valid worktree is detected, it MUST become the **implementation directory** for all subsequent operations (tracker creation, file edits, test runs, verification artifacts).

§3.1.3 STRUCT Awareness

- **FR-1.8** Before parsing the spec, the skill SHOULD check for `.spec-tracker-*.md` files in the specification's directory.
- **FR-1.9** If a spec tracker with a `## Pending Structural Changes` section is found, the skill MUST warn the user and ask whether to proceed or wait for resolution.
- **FR-1.10** If the user proceeds despite pending structural changes, the skill MUST note the pending issues in the tracker's Implementation Log.

§3.1.4 Specification Parsing

- **FR-1.11** The skill MUST detect whether the specification is single-file or multi-file. Detection MUST check for `<!-- EXPANDED:` markers in the master document or a `sections/` directory alongside it.

- **FR-1.12** The detected spec type MUST be recorded in the tracker as `**Spec Type**: single-file` or `**Spec Type**: multi-file`.

Single-File Specs

- **FR-1.13** For single-file specs, the skill MUST read the entire specification document.
- **FR-1.14** The skill MUST identify the document's section structure (patterns such as `## Section N`, `### N.M`, `§N.M`, numbered headings).
- **FR-1.15** The skill MUST extract each discrete requirement with its section reference.

Multi-File Specs

- **FR-1.16** For multi-file specs, the skill MUST NOT read all section files into main context.
- **FR-1.17** The skill MUST build a structural index by:
  1. Reading the master spec's document map / table of contents only.
  2. Running `wc -c` on all section files to compute byte counts.
  3. Estimating tokens as `estimated_tokens = bytes / 4`.
- **FR-1.18** The skill MUST handle sub-file splitting (letter suffixes such as `02a-core-model.md`, `02b-core-relations.md`) by grouping sub-files under their parent section number.
- **FR-1.19** The skill MUST read only section headings and requirement identifiers (MUST/SHOULD/COULD statements) from each section file — not full prose.
- **FR-1.20** The structural index MUST be stored in the tracker for use as a spec baseline in future sessions.

§3.1.5 Tracker Creation

- **FR-1.21** The skill MUST create a tracker file in the implementation directory (worktree or current working directory).
- **FR-1.22** The tracker MUST be named `.impl-tracker-<spec-basename>.md`, where `<spec-basename>` is the filename of the spec without its directory path or final `.md` extension. Only the final `.md` extension is stripped (e.g., `api.v2.spec.md` yields basename `api.v2.spec`).
- **FR-1.23** The tracker MUST conform to the template defined in `references/tracker-format.md`, including at minimum:
  - Metadata fields: spec path, creation date, TDD mode, spec type, spec baseline, worktree path, branch.
  - Recovery Instructions section with self-contained resumption guidance.
  - Requirements Matrix with columns: Section, Requirement, Priority, Status, Implementation, Tests.
  - Structural Index (multi-file specs only).
  - Known Gaps section.
  - Deviations from Spec section.
  - Implementation Log section.
- **FR-1.24** The tracker MUST include machine-readable comment fields (`<!-- SPEC_PATH: ... -->`, `<!-- TDD_MODE: ... -->`, etc.) for programmatic parsing.

§3.1.6 Task Creation

- **FR-1.25** The skill MUST create tasks via `TaskCreate` for each major requirement or group of related requirements.
- **FR-1.26** Each task's subject MUST include section references (e.g., `Implement merge detection workflow (§2.4, §10.2)`).
- **FR-1.27** Each task's description MUST include:
  - The spec section references being addressed.

  - The specific requirements extracted from those sections.
  - A reminder to re-read the referenced sections before starting.

§3.1.7 TDD Mode Determination

- **FR-1.28** The skill MUST determine TDD mode using the following preference lookup chain:
  1. Project-level: `.impl-preferences.md` in the project directory.
  2. Global: `~/.claude/.impl-preferences.md`.
  3. Built-in default: `on`.
- **FR-1.29** If the preference is `on` or `off`, the skill MUST record it without prompting the user.
- **FR-1.30** If the preference is `ask`, the skill MUST present the TDD/Standard workflow choice to the user as part of the plan presentation.
- **FR-1.31** The determined mode MUST be recorded in the tracker's `**TDD Mode**:` field as either `on` or `off`.

§3.1.8 Plan Presentation and Approval

- **FR-1.32** Before proceeding to implementation, the skill MUST present the plan to the user, including:
  1. A summary of the specification structure.
  2. The requirements matrix from the tracker.
  3. The proposed task breakdown.
  4. The implementation workflow (TDD or Standard).
  5. Any questions about ambiguous requirements.
- **FR-1.33** The skill MUST NOT begin implementation until the user approves the plan.
- **FR-1.34** If TDD preference was `ask`, the user's workflow choice MUST be captured during plan approval and recorded in the tracker.

---

§3.2 Phase 2: Implementation

Phase 2 executes the implementation plan created in Phase 1, delegating work to sub-agents while the orchestrator maintains tracker state and enforces quality gates. Two workflows exist depending on TDD mode.

> **Artifact directory rationale**: Implementation artifacts (`summary.json`, `compliance.json`, `fix-summary.json`) are stored in `.impl-work/<spec-name>/` because they are per-task working files produced during implementation. Verification artifacts (JSON fragments, assembled reports) are stored in `.impl-verification/<spec-name>/` because they are per-run diagnostic outputs produced during verification. This separation keeps working state distinct from audit outputs.

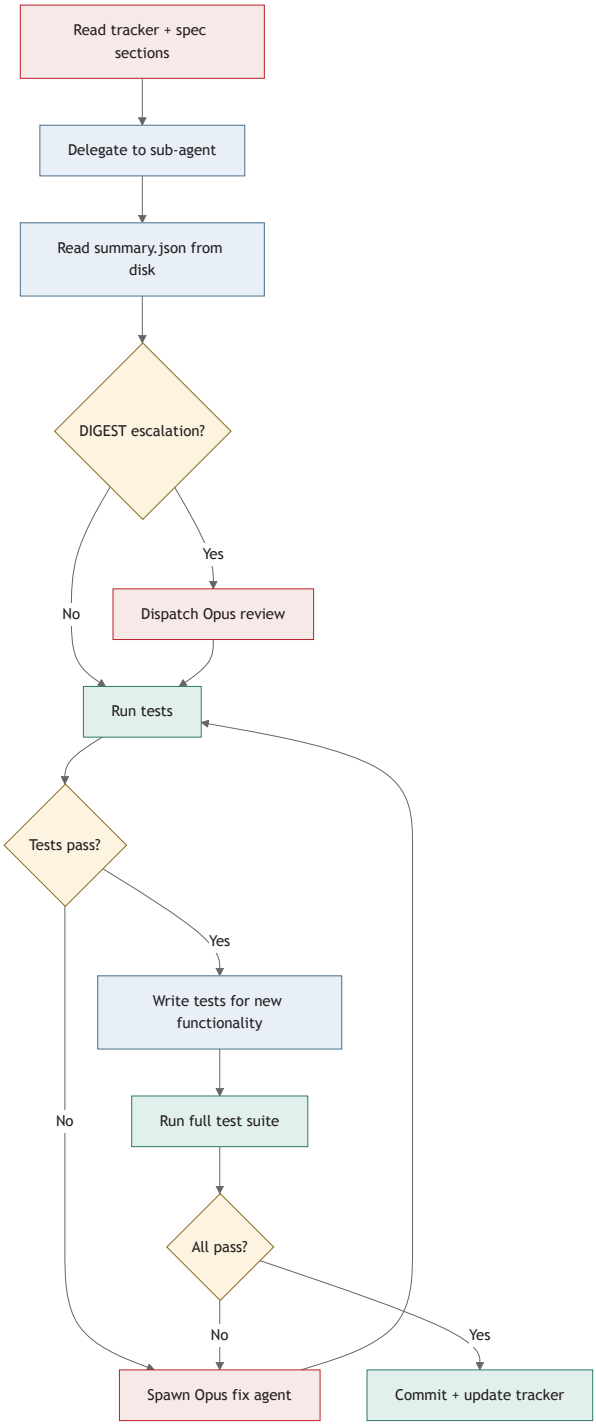
§3.2.1 Pre-Implementation Checks

These checks apply to both Standard and TDD workflows.

- **FR-2.1** The skill MUST verify that a tracker file (`.impl-tracker-*.md`) exists in the implementation directory before writing any implementation code.
- **FR-2.2** The skill MUST verify that the tracker has a populated Requirements Matrix.
- **FR-2.3** The skill MUST verify that tasks have been created via `TaskCreate`.
- **FR-2.4** If any of FR-2.1 through FR-2.3 fail, the skill MUST stop and require completion of Phase 1 before proceeding. This is a hard gate.

§3.2.2 Standard Workflow (TDD Off)

The Standard workflow implements code first, then writes tests afterward.



Step 1: Context Preparation

- **FR-2.5** Before delegating any task, the orchestrator MUST read the tracker file to obtain section references.
- **FR-2.6** The orchestrator MUST re-read the relevant spec sections from the original document (single-file) or confirm which section files the sub-agent will read (multi-file). This is mandatory, not optional.
- **FR-2.7** The orchestrator MUST identify relevant existing code files the sub-agent will need.

Step 2: Sub-Agent Delegation

- **FR-2.8** Before dispatching a sub-agent, the skill MUST clear previous completion markers:

```
mkdir -p <impl-dir>/.impl-work/<spec-name>/ && rm -f <impl-dir>/.impl-work/<spec-name>/summary.done
```

- **FR-2.8a** Implementation sub-agents MUST write `summary.done` as a completion marker after writing `summary.json`, analogous to verification agents writing `.done` markers (FR-3.20). The orchestrator detects sub-agent completion by checking for this marker.
- **FR-2.9** The skill MUST delegate implementation to a sub-agent using the appropriate prompt template:
  - Single-file specs: `prompts/implement-single-file.md`.
  - Multi-file specs: `prompts/implement-multi-file.md`.
- **FR-2.10** Model selection MUST follow the routing rules in `references/sub-agent-strategy.md`:
  - Straightforward tasks: `haiku` or `sonnet`.
  - Moderate/complex tasks: `opus`.
  - For multi-file specs, size-based routing: <5k tokens per section to sonnet (group 2-3), 5k-20k to sonnet (1 each), >20k to opus (1 each).
  - Default: if no routing rule matches, use `sonnet`.

Step 3: Validation

- **FR-2.11** After the sub-agent completes, the orchestrator MUST read the structured summary from `<impl-dir>/.impl-work/<spec-name>/summary.json`. The orchestrator MUST NOT re-analyse the agent's conversational output.
- **FR-2.12** The orchestrator MUST check the `concerns` field — if non-empty, the orchestrator MUST investigate.
- **FR-2.13** The orchestrator MUST check the `status` field — if not `complete`, the orchestrator MUST investigate.
- **FR-2.14** For sonnet sub-agents on multi-file specs, the orchestrator MUST read the `digest` field and check signals against the complexity category table in `references/sub-agent-strategy.md` (categories: algorithms, state machines, permission/auth, complex business rules, cross-cutting).
- **FR-2.15** If a DIGEST signal matches any complexity category, Opus review of the sonnet's code changes MUST be dispatched. This escalation is mandatory, not discretionary.
- **FR-2.16** The orchestrator MUST run the test suite (full suite or at minimum the relevant test files) after sub-agent completion.
- **FR-2.17** If tests fail, the skill MUST spawn an Opus fix agent using `prompts/fix-issue.md`, then re-run tests. Note: the skill does not impose a hard iteration limit on test-fix cycles. In practice, Claude's natural behaviour acts as a circuit breaker — if a fix is not converging after 2-3 attempts, the orchestrator should escalate to the user rather than continuing to loop.

Step 4: Test Writing

- **FR-2.18** After implementation passes tests, the skill MUST delegate test writing for new functionality using the prompt template at `prompts/write-tests.md`.
- **FR-2.19** Previous completion markers MUST be cleared before dispatching the test-writing sub-agent.
- **FR-2.20** After the test sub-agent completes, the full test suite MUST be run to confirm both new and existing tests pass. A task MUST NOT be marked `complete` until both new and existing tests pass (equivalent to TDD FR-2.50).

Step 5: Commit

- **FR-2.21** If the project is a git repository, the skill SHOULD create an atomic commit for each task's changes (implementation + tests).
- **FR-2.22** The commit message SHOULD reference the spec section (e.g., `feat: implement merge detection (§2.4)`).
- **FR-2.23** Only files changed by the current task SHOULD be staged.

Step 6: Tracker Update

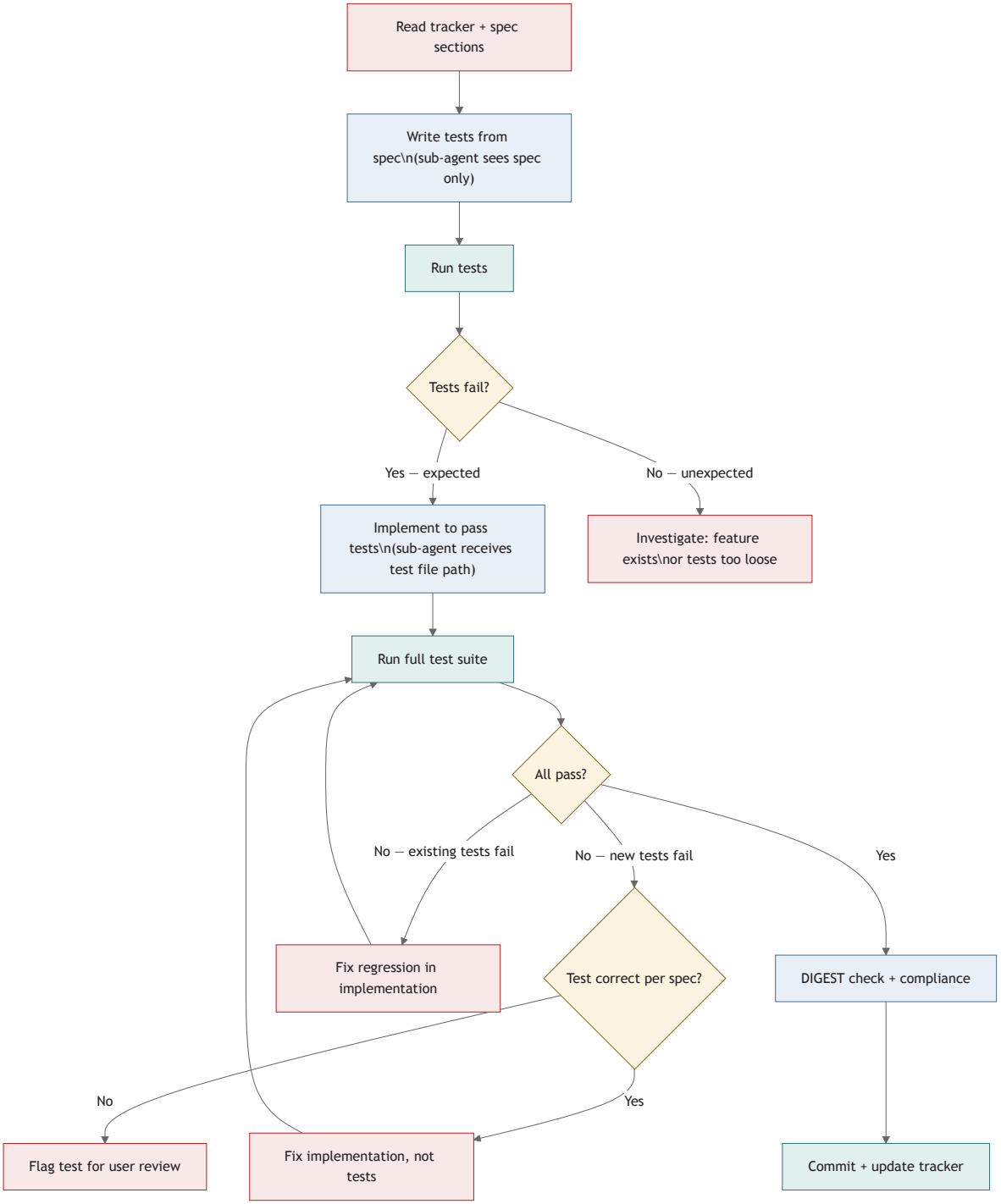
- **FR-2.24** The skill MUST update the tracker's Requirements Matrix with status, implementation file:line references, and test file references. Status values on task completion: `complete` on success, `partial` on partial success (some requirements met but gaps remain), `blocked` on failure or inability to proceed.
- **FR-2.25** The skill MUST add an entry to the Implementation Log.
- **FR-2.26** The skill MUST update the task status via `TaskUpdate`.
- **FR-2.27** A task MUST NOT be marked `complete` if any of the following are true:
  - Tests are failing.
  - Tests have not been run.
  - New functionality has no tests.
  - There are linting or type errors.

Step 7: Spec Compliance Check

- **FR-2.28** The skill SHOULD run an optional spec compliance check for non-trivial tasks using `prompts/spec-compliance-check.md`.
- **FR-2.29** The compliance check result MUST be read from `<impl-dir>/.impl-work/<spec-name>/compliance.json`, not from conversational output.

§3.2.3 TDD Workflow (TDD On)

The TDD workflow writes tests first from the specification, then implements code to make them pass. This exploits context isolation between agents: the test-writing agent interprets the spec independently of any implementation.



Step 1: Context Preparation

- **FR-2.30** Context preparation for TDD follows the same requirements as Standard workflow (FR-2.5 through FR-2.7).
- **FR-2.31** The orchestrator MUST additionally identify existing test conventions and test file locations.

Step 2: Write Tests First

- **FR-2.32** Previous completion markers MUST be cleared before dispatching the test-writing sub-agent.
- **FR-2.33** The skill MUST delegate test writing using the prompt template at `prompts/tdd-write-tests.md`.
- **FR-2.34** The test-writing sub-agent MUST work purely from the specification. It MUST NOT see any implementation code. This is the critical context isolation that ensures tests verify spec requirements, not implementation behaviour.
- **FR-2.35** After the test-writing agent completes, the orchestrator MUST read the summary from `<impl-dir>/.impl-work/<spec-name>/summary.json`.

Step 3: Confirm Test Failures

- **FR-2.36** The skill MUST run the test suite after test writing. New tests SHOULD fail, confirming they check something real.
- **FR-2.37** If new tests pass unexpectedly, the skill MUST investigate — the feature may already exist or the tests may be too loose.
- **FR-2.38** If tests error due to import or syntax issues, the skill MUST fix the test setup (imports, fixtures, file paths) but MUST NOT modify test assertions.

Step 4: Implement to Pass

- **FR-2.39** Previous completion markers MUST be cleared before dispatching the implementation sub-agent.
- **FR-2.40** The skill MUST delegate implementation using `prompts/tdd-implement.md`.
- **FR-2.41** The implementation sub-agent MUST receive the test file path as acceptance criteria.

Step 5: Validate

- **FR-2.42** The orchestrator MUST read the structured summary from disk (FR-2.11 pattern).
- **FR-2.43** DIGEST escalation rules MUST apply (FR-2.14, FR-2.15). Note: DIGEST escalation applies to all sonnet sub-agents regardless of spec type, but single-file specs use the single-file prompt which does not include a DIGEST field, so in practice DIGEST escalation only triggers for multi-file specs.
- **FR-2.44** The full test suite MUST be run. Both new tests and existing tests MUST pass.
- **FR-2.45** If new tests still fail, the skill MUST fix the implementation, not the tests.
- **FR-2.46** If a test appears genuinely wrong (misinterprets the spec), the skill MUST flag it for user review rather than silently modifying it.
- **FR-2.47** If existing tests break, the skill MUST fix the regression in the implementation.

Step 6: Critical Rule — Test Integrity

- **FR-2.48** The skill MUST NEVER silently modify tests to match a wrong implementation. When tests disagree with implementation, the spec is the arbiter. If the test correctly reflects the spec, the implementation MUST be fixed. If the test misinterprets the spec, it MUST be flagged for user review.

Step 7: Commit and Tracker Update

- **FR-2.49** Commit and tracker update follow the same requirements as Standard workflow (FR-2.21 through FR-2.27).
- **FR-2.50** A task MUST NOT be marked `complete` until both new and existing tests pass.

§3.2.4 Standard vs TDD Workflow Comparison

Step	Standard Workflow	TDD Workflow	Notes
Task start	Read tracker + spec sections (FR-2.5–2.7)	Same + identify test conventions (FR-2.30–2.31)	TDD adds test location discovery
Test writing	After implementation (FR-2.18)	Before implementation (FR-2.33–2.34)	TDD tests written from spec only — no implementation code visible
Implementation	Sub-agent implements from spec (FR-2.9)	Sub-agent implements to pass tests (FR-2.40–2.41)	TDD sub-agent receives test file path as acceptance criteria
Test running	Run after implementation + after test writing (FR-2.16, FR-2.20)	Run after test writing (expect failures), then after implementation (FR-2.36, FR-2.44)	TDD confirms tests fail first
Completion gate	All tests pass, new functionality has tests (FR-2.27)	Both new and existing tests pass (FR-2.50)	Equivalent gates, TDD makes it explicit
DIGEST check	Sonnet sub-agents on multi-file specs (FR-2.14)	Same rules apply (FR-2.43)	DIGEST only in multi-file prompts

§3.2.5 Sub-Agent Issue Handling

- **FR-2.51** If a sub-agent's implementation has gaps or errors in either workflow, the skill MUST use `prompts/fix-issue.md` to spawn an Opus fix agent.
- **FR-2.52** Fix agents MUST always use `model: "opus"`. This is non-negotiable.

---

§3.3 Phase 3: Verification (`/implement verify`)

Phase 3 performs systematic, per-requirement verification using parallel sub-agents that produce structured JSON fragments, assembled into a report by deterministic Python tooling. Verification MUST always use Opus-class models for sub-agents.

§3.3.1 Pre-Verification Gate

- **FR-3.1** Before any spec verification, the skill MUST run the test suite. If tests fail, the skill MUST fix them before proceeding. Verification of non-functional code is meaningless.
- **FR-3.2** The skill MUST run linting and type checking if configured (e.g., `mypy`, `flake8`, `eslint`, `tsc`). Errors MUST be fixed before proceeding.
- **FR-3.3** The skill MUST verify that the code compiles/runs (compiled languages build without errors; interpreted languages pass import/load checks; web apps can start the dev server).
- **FR-3.4** If any of FR-3.1 through FR-3.3 fail, the skill MUST NOT proceed with verification. This is a hard gate.

§3.3.2 Tracker and Worktree Resolution

- **FR-3.5** If a spec-name is provided, the skill MUST look for `.impl-tracker-<spec-name>.md`.

- **FR-3.6** If no spec-name is provided and exactly one tracker exists, the skill MUST use it. If multiple exist, the skill MUST present the list and ask the user to choose. If none exist, the skill MUST inform the user.
- **FR-3.7** After finding the tracker, the skill MUST check its `**Worktree**` field. If not `none`, the worktree path MUST be validated and used as the implementation directory for all subsequent steps.

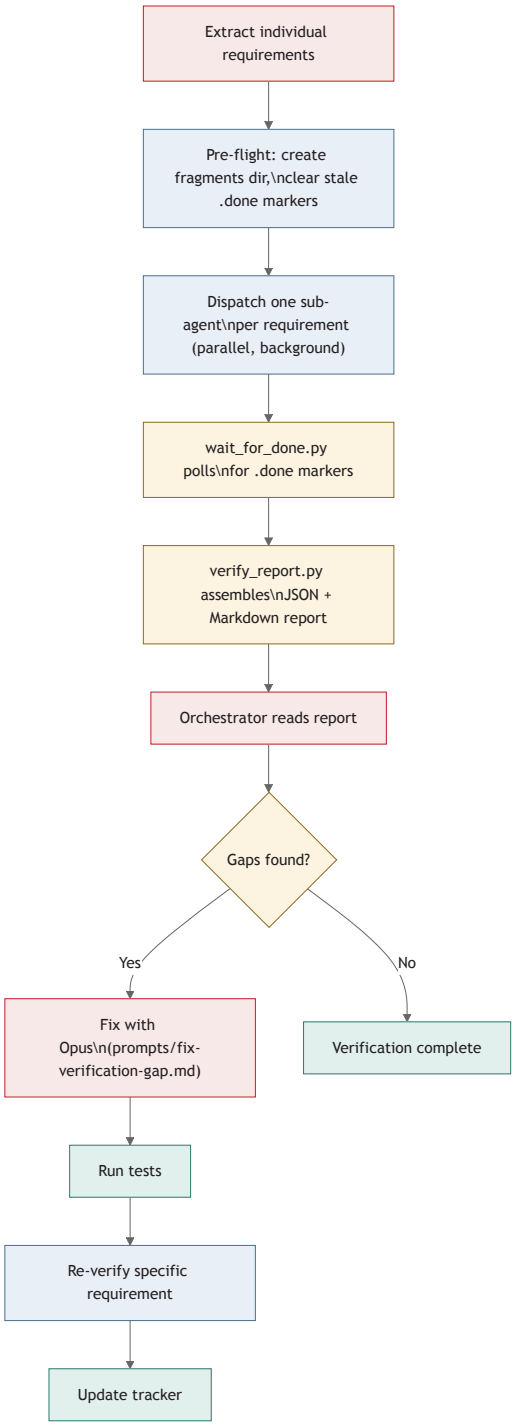
§3.3.3 Verification Plan

- **FR-3.8** In the main conversation, the skill MUST read only the tracker and the spec's structure/table of contents. Full spec sections and implementation files MUST NOT be read into the main conversation context.
- **FR-3.9** The skill MUST check for previous verification reports at `<impl-dir>/.impl-verification/<spec-name>/verify-*.json`. If found, the most recent report MUST be read, triggering re-verification mode (see §3.3.7).

§3.3.4 Requirement Extraction

- **FR-3.10** The skill MUST extract individual requirements from each spec section — the specific MUST/SHOULD/COULD statements or concrete behavioural expectations, not section headings.
- **FR-3.11** The extraction MUST distinguish between topic areas (e.g., "§2.1 Quick Capture") and individual requirements within those topics. A section with 15 subsections SHOULD produce 30-60+ individual requirements.
- **FR-3.12** For each requirement, the skill MUST record: the section reference (§N.M), a one-line summary, and any implementation hints from the tracker (file:line references).
- **FR-3.13** After extracting requirements from a section, the skill SHOULD release the section content from context, retaining only the flat requirement list.

§3.3.5 Verification Dispatch



- **FR-3.14** Before dispatching sub-agents, the skill MUST create the fragments directory and clear any stale `.done` markers:

```
mkdir -p <impl-dir>/.impl-verification/<spec-name>/fragments/ && rm -f
    ↪ <impl-dir>/.impl-verification/<spec-name>/fragments/*.done
```

- **FR-3.15** During initial verification, the skill MUST dispatch exactly one sub-agent per requirement. Requirements MUST NOT be batched during initial verification. Exception: re-verification spot-checks (FR-3.32) MAY batch 5-10 passed items into a single agent for efficiency.
- **FR-3.16** Verification sub-agents MUST use `model: "opus"`.
- **FR-3.17** Verification sub-agents MUST be dispatched with `run_in_background: true`. Note: the skill does not impose a concurrency cap on verification agents. For large specs (100+ requirements), all agents are dispatched in parallel with the expectation that API-level rate limiting provides backpressure. The `wait_for_done.py` timeout (600s default) acts as the only bound.
- **FR-3.18** Each sub-agent MUST use the prompt template at `prompts/verify-requirement.md`.
- **FR-3.19** Each sub-agent MUST write a JSON fragment and a `.done` marker to `<impl-dir>/.impl-verification/<spec-name>/fra`
- **FR-3.20** The orchestrator MUST NOT call `TaskOutput` on verification agents. Completion MUST be detected via `.done` markers only.

§3.3.6 Report Assembly

- **FR-3.21** The skill MUST wait for all sub-agents to complete by running:

```
"$IMPL_PYTHON" "$IMPL_TOOLS_DIR/wait_for_done.py" --dir <fragments-dir> --count <N>
```

- **FR-3.22** The skill MUST assemble the report using the deterministic Python tool:

```
"$IMPL_PYTHON" "$IMPL_TOOLS_DIR/verify_report.py" \
  --fragments-dir <fragments-dir> \
  --spec-path <spec-path> \
  --impl-path <impl-dir> \
  --project-name "<spec-name>" \
  --output <impl-dir>/.impl-verification/<spec-name>/verify-<date>.json
```

- **FR-3.23** The assembly tool MUST produce both a JSON report and a Markdown report. The Markdown format is defined by `tools/verification_schema.py:render_markdown()` — the orchestrator MUST NOT write report markdown manually.
- **FR-3.24** The skill MUST present a summary to the user including: requirements count (X of Y implemented), test coverage (A of B), implementation rate, and the top critical gaps.

§3.3.7 Gap Fixing

- **FR-3.25** Gap fixes MUST always use `model: "opus"`. This is non-negotiable.
- **FR-3.26** Gap fixes MUST use the prompt template at `prompts/fix-verification-gap.md`.
- **FR-3.27** After each fix, the orchestrator MUST:
  1. Read the fix summary from `<impl-dir>/.impl-work/<spec-name>/fix-summary.json`.
  2. Run tests to confirm the fix works.
  3. Re-verify that specific requirement (single sub-agent, background, same pattern as initial dispatch).
  4. Wait for `.done`, then read the updated fragment.
  5. Update the tracker.
- **FR-3.28** The fix-verify cycle MUST repeat until all gaps are resolved or explicitly documented. Note: no hard iteration limit is imposed on fix-verify cycles. If gaps persist after 2-3 fix attempts, the orchestrator should document remaining gaps and escalate to the user.

§3.3.8 Re-Verification Mode

- **FR-3.29** When a previous verification report exists, the skill MUST enter re-verification mode and ask the user to choose between:
  1. **Re-verify from where we left off** — check only open V-items plus spot-check for regressions.
  2. **Full re-verification from scratch** — re-audit all requirements, carrying forward V-item IDs.

Re-Verify From Where We Left Off

- **FR-3.30** The skill MUST read the most recent report and extract open V-items and the ID counter.
- **FR-3.31** Open items (Partial/Not Implemented, or test coverage Partial/None) MUST be re-verified using `prompts/reverify-requirement.md`.
- **FR-3.32** Passed items MUST receive a lightweight spot-check (cluster 5-10 into one agent).
- **FR-3.33** The skill SHOULD check for new requirements if the spec was updated since the last verification.

Full Re-Verification From Scratch

- **FR-3.34** The full initial verification flow MUST be executed.
- **FR-3.35** Findings MUST be matched to previous V-items by section reference. Fallback: when section references change between verification runs (e.g., spec restructuring), V-items are matched by requirement text similarity. Items that cannot be matched are treated as new.
- **FR-3.36** Existing V-item IDs MUST be reused.
- **FR-3.37** Resolution status MUST be included for previously flagged items.

V-Item Lifecycle

- **FR-3.38** V-item IDs MUST be permanent — once assigned, they persist across all re-verification runs.
- **FR-3.39** Resolution statuses MUST use the following values:
  - `fixed` — fully resolved.
  - `partially_fixed` — progress made, not fully resolved.
  - `not_fixed` — no meaningful progress.
  - `regressed` — previously fixed, now broken again. Human-readable display (e.g., in Markdown reports) MAY use title case (e.g., "Fixed", "Partially Fixed") but the canonical stored values MUST be lowercase underscore format.

§3.3.9 Re-Verification Report Assembly

- **FR-3.40** For re-verification, the report assembly command MUST include the `--previous` flag pointing to the previous report JSON.
- **FR-3.41** The assembled report MUST include delta information showing how V-items changed between runs.

§3.3.10 Context Efficiency Rules

- **FR-3.42** The main conversation MUST read only the tracker and spec structure during verification.
- **FR-3.43** Spec requirement text MUST be passed directly in sub-agent prompts, not loaded into main context.
- **FR-3.44** Sub-agents MUST read implementation files themselves.
- **FR-3.45** Sub-agent output MUST be limited to structured findings (JSON fragments).

---

§3.4 Phase 4: Status (`/implement status`)

Phase 4 is a read-only inspection of implementation progress. It MUST NOT modify any files.

§3.4.1 Tracker Discovery

- **FR-4.1** If a spec-name is provided, the skill MUST look for `.impl-tracker-<spec-name>.md`.
- **FR-4.2** If no spec-name is provided and exactly one tracker exists, the skill MUST use it. If multiple exist, the skill MUST present the list and ask. If none exist, the skill MUST inform the user.

§3.4.2 Status Display

- **FR-4.3** The skill MUST read the tracker file and the current task list.
- **FR-4.4** The skill MUST present a summary containing:
  1. Overall progress — X of Y requirements complete.
  2. Current task being worked on.
  3. Blockers or gaps discovered.
  4. Sections not yet started.
- **FR-4.5** The status display MUST be read-only. The skill MUST NOT modify the tracker, tasks, or any implementation files.

---

§3.5 Phase 5: Continue (`/implement continue`)

Phase 5 resumes implementation from a previous session, re-establishing context from the persistent tracker and validating that the environment is still consistent.

§3.5.1 Tracker Discovery

- **FR-5.1** Tracker discovery follows the same rules as Phase 4 (FR-4.1, FR-4.2).

§3.5.2 Worktree Re-Validation

- **FR-5.2** If the tracker's `**Worktree**` field is not `none`, the skill MUST:
  1. Verify the worktree path still exists on disk.
  2. Verify it still appears in `git worktree list`.
  3. If `**Branch**` is not `none`, verify the worktree is on the expected branch.
- **FR-5.3** If worktree validation fails, the skill MUST warn the user and offer three options: re-create the worktree, work in the current directory instead, or abort.
- **FR-5.4** If validation passes, the worktree path MUST be set as the implementation directory.

§3.5.3 Spec Freshness Check

- **FR-5.5** For multi-file specs, the skill MUST re-run `wc -c` on section files and compare against the stored Structural Index. The skill MUST detect:
  - **New files**: files present on disk but not in the index.
  - **Removed files**: files in the index but no longer on disk.
  - **Size changes**: any file whose byte count changed by >20% from the stored value.
  - **Sub-split patterns**: a previously single file now has letter-suffix variants.
- **FR-5.6** For single-file specs, the skill MUST compare the file's modification time against the tracker's `**Spec Baseline**` date.

Note (precision limitation): The Spec Baseline field stores a date (YYYY-MM-DD), while file modification time has second-level precision. Same-day edits will not trigger the freshness warning. This is an accepted limitation — single-file freshness uses mtime-vs-date comparison, so changes made on the same calendar day as the baseline date are not detected.

- **FR-5.7** The skill SHOULD check for `.spec-tracker-*.md` files with `## Pending Structural Changes` (STRUCT check).
- **FR-5.8** When no changes are detected, the skill MUST proceed silently.
- **FR-5.9** When changes are detected, the skill MUST present three options:

Option 1: Re-Scan Affected Sections

- **FR-5.10** The skill MUST re-read only the changed/new section files and compare requirements against the existing Requirements Matrix.
- **FR-5.11** New requirements MUST be added as `pending` rows.
- **FR-5.12** Removed requirements MUST be marked `n/a` with a note.
- **FR-5.13** Changed requirements MUST be flagged as `pending` with a note indicating the requirement text changed since the last scan.
- **FR-5.14** The Structural Index and Spec Baseline date MUST be updated.
- **FR-5.15** New tasks MUST be created for added requirements.

Option 2: Proceed As-Is

- **FR-5.16** The skill MUST log the detected changes in the Implementation Log with details (new files, removed files, size changes) and note that the user chose to proceed without re-scanning.

Option 3: Full Re-Plan

- **FR-5.17** The skill MUST archive the current tracker with a date suffix.
- **FR-5.18** The skill MUST re-run Phase 1 from scratch, carrying forward completed work, known gaps, deviations, and Implementation Log entries from the archived tracker. The full Implementation Log history is carried forward — entries are NOT summarised (see §8.4.2).

§3.5.4 Resuming Implementation

- **FR-5.19** The skill MUST read the tracker's `**TDD Mode**:` field and use the corresponding workflow (TDD or Standard).
- **FR-5.20** The skill MUST read the task list and identify the next pending task.
- **FR-5.21** The skill MUST re-read the relevant spec sections before continuing implementation.
- **FR-5.22** The skill MUST resume implementation using the appropriate Phase 2 workflow.

---

§3.6 Configuration (`/implement config`)

The configuration subsystem manages workflow preferences at two scopes: project-level and global.

§3.6.1 Preference Files

- **FR-6.1** Project-level preferences MUST be stored in `.impl-preferences.md` in the project directory.
- **FR-6.2** Global preferences MUST be stored in `~/.claude/.impl-preferences.md`.
- **FR-6.3** Preference lookup MUST follow the chain: project file, then global file, then built-in defaults.
- **FR-6.4** Project-level preferences MUST override global preferences.

- **FR-6.4a** The `.impl-preferences.md` file MUST use the following format: a markdown file with a `# Implementation Preferences` heading followed by a section heading per preference category (e.g., `## Workflow`) containing bullet list items in `**key**: value` format (e.g., `- **tdd-mode**: on`). The lookup chain is: project `.impl-preferences.md` → `~/.claude/.impl-preferences.md` → built-in default.

§3.6.2 Config Commands

- **FR-6.5** `/implement config` (no arguments) MUST display the current effective preferences, showing the resolved value and its source (project, global, or default).
- **FR-6.6** `/implement config tdd on|off|ask` MUST update the TDD preference at project level (`.impl-preferences.md`).
- **FR-6.7** `/implement config --global tdd on|off|ask` MUST update the TDD preference at global level (`~/.claude/.impl-preferences.md`).
- **FR-6.8** When setting a preference, the skill MUST:
  1. Read the existing preferences file (or create it if it does not exist).
  2. Update the specified value.
  3. Write the file back.
  4. Confirm the change to the user.

§3.6.3 Available Preferences

- **FR-6.9** The `tdd-mode` preference MUST accept three values: `on`, `off`, `ask`.
  - `on` — always use the TDD workflow.
  - `off` — always use the standard workflow.
  - `ask` — prompt during Phase 1 planning for user choice.
- **FR-6.10** The built-in default for `tdd-mode` MUST be `on`.

---

§3.7 List (`/implement list`)

- **FR-7.1** The skill MUST find all `.impl-tracker-*.md` files in the current directory and any known worktree paths.
- **FR-7.2** The skill MUST present a summary table containing at minimum: tracker name, spec path, overall status, progress (X of Y requirements complete), and last updated date.
- **FR-7.3** If no trackers are found, the skill MUST inform the user that no active implementations exist.

---

§3.8 Implicit Activation

The skill MAY be activated without an explicit `/implement` invocation when the user's message mentions tracker files, implementation gaps, or spec verification.

§3.8.1 Detection

- **FR-8.1** The skill SHOULD detect when the user's message references `.impl-tracker` files, implementation gaps, gap-fixing, or spec verification.

§3.8.2 Consent Requirement

- **FR-8.2** On implicit activation, the skill MUST NOT silently take over. The skill MUST ask the user if they would like to use the implementation skill.
- **FR-8.3** If the user agrees, the skill MUST read the tracker to determine the appropriate phase and proceed.
- **FR-8.4** If the user declines, the skill MUST assist normally without invoking skill-specific behaviour.

§3.8.3 Announcement

- **FR-8.5** When implicitly activated (and the user consents), the skill MUST announce: `"It looks like you're working with an`

---

§3.9 Command Routing

This subsection specifies how the skill interprets arguments to determine which phase to enter.

- **FR-9.1** If arguments contain a file path, the skill MUST enter Phase 1 (Planning). A path argument is treated as a file path if: (a) it contains `/` AND the path exists on disk, OR (b) it ends in `.md` or `.txt` AND the file exists. Non-existent paths MUST NOT be routed as file paths — the skill SHOULD inform the user that the file was not found.
- **FR-9.2** If arguments start with `status`: the skill MUST enter Phase 4 (Status).
- **FR-9.3** If arguments start with `verify`: the skill MUST enter Phase 3 (Verification).
- **FR-9.4** If arguments start with `continue`: the skill MUST enter Phase 5 (Continue).
- **FR-9.5** If arguments are `list`: the skill MUST list all trackers (§3.7).
- **FR-9.6** If arguments start with `config`: the skill MUST handle preferences (§3.6).
- **FR-9.7** If arguments are empty, the skill MUST follow the empty arguments procedure:
  1. Check for existing trackers in the current directory and known worktree paths.
  2. If exactly one tracker exists: offer to continue implementation.
  3. If multiple trackers exist: show list and ask which to continue.
  4. If no trackers exist: search for spec documents (`**/*spec*.md`, `**/*requirements*.md`), present candidates, or ask for a path.
  5. Once a spec path is identified: always enter Phase 1 (Planning).

§3.9.1 Announcement on Activation

- **FR-9.8** On explicit invocation, the skill MUST announce its action, e.g.: `"I'm using the implement skill to [plan/implement/v`

§4 Data Model & Artifacts

This section specifies every file artifact that the `/implement` skill creates, reads, or manages. Together these artifacts form the persistent substrate that makes the skill resilient to context compaction, session restarts, and model switches. Understanding the data model is prerequisite to understanding the workflow: each phase reads from and writes to a specific set of these artifacts.

The section is organised in two logical parts:

- **Artifact Schemas** (§4.1–§4.6): File formats, naming conventions, field definitions, and validation rules for every persistent and transient artifact.
- **State Machines and Lifecycle** (§4.7–§4.8): Advisory lifecycle documentation describing how artifacts transition through states and how files are managed over time.

---

§4.1 Implementation Tracker

File pattern: `.impl-tracker-<spec-name>.md` **Location**: Project root (or worktree root, if a worktree is active) **Created by**: Phase 1 (Planning) **Read by**: All phases; self-recovering after context compaction **Committed to git**: MUST be committed — it is the persistent bridge across sessions

The implementation tracker is the central persistent artifact of the skill. It encodes enough context about the specification and the current implementation state that a fresh Claude session can reconstruct its understanding and resume work accurately without access to any prior conversation history.

§4.1.1 Naming Convention

The tracker filename is derived from the specification path by taking the file stem and appending it to the prefix `.impl-tracker-`. This scheme allows multiple trackers to coexist in a single project.

Spec Path	Tracker Filename
`docs/billing-spec.md`	`.impl-tracker-billing-spec.md`
`notification-system.md`	`.impl-tracker-notification-system.md`
`../specs/auth-flow.md`	`.impl-tracker-auth-flow.md`

When working inside a git worktree, the tracker MUST be placed at the worktree root rather than the main repository root, so that each worktree maintains its own independent implementation state.

§4.1.2 Header Fields

The tracker begins with a set of structured header fields. These fields MUST appear at the top of the file and MUST use **bold-label markdown format** so they are unambiguous after context compaction.

Bold-label format syntax: `**Field**: value` — a bold label (enclosed in `**`), followed by a colon, a space, and the value. Example: `**Status**: In Progress`.

Field	Values	Description
`Specification`	File path (string)	Relative or absolute path to the source spec
`Created`	`YYYY-MM-DD`	Date the tracker was first created

Field	Values	Description
`Last Updated`	`YYYY-MM-DD`	Date of the most recent update
`Status`	`Planning` / `In Progress` / `Verification` / `Complete`	Overall tracker lifecycle status
`TDD Mode`	`on` / `off`	Whether TDD workflow is active for this implementation
`Spec Type`	`single-file` / `multi-file`	Whether the spec uses breakout section files
`Spec Baseline`	`YYYY-MM-DD`	Date the structural index baseline was captured. Written once during Phase 1 planning; refreshed only during `/implement continue` when structural index changes are detected
`Worktree`	Absolute path or `none`	Active git worktree path, or `none`
`Branch`	Branch name or `none`	Active branch, or `none`
`Tracker Format`	Version string (e.g. `1.0`)	Format version of the tracker layout; increment when tracker format changes

The header fields MUST also be encoded as HTML comments immediately after the markdown header for machine-readable access:

```
<!-- SPEC_PATH: path/to/spec.md -->
<!-- TDD_MODE: on|off -->
<!-- SPEC_TYPE: single-file|multi-file -->
<!-- SPEC_BASELINE: YYYY-MM-DD -->
<!-- WORKTREE: /absolute/path/or/none -->
<!-- BRANCH: branch-name-or-none -->
<!-- LAST_SECTION: §4.7 -->
<!-- COMPLETE_COUNT: 18 -->
<!-- PARTIAL_COUNT: 4 -->
<!-- GAP_COUNT: 2 -->
<!-- PENDING_COUNT: 0 -->
```

The convenience fields `LAST_SECTION`, `COMPLETE_COUNT`, `PARTIAL_COUNT`, `GAP_COUNT`, and `PENDING_COUNT` are machine-readable summary counters maintained for use by scripts and tooling. They are updated whenever the tracker is refreshed (e.g. after a verification run or implementation batch). `LAST_SECTION` records the last spec section processed; the count fields track how many requirements are in each status category.

Invariant: The sum `COMPLETE_COUNT + PARTIAL_COUNT + PENDING_COUNT` plus any `blocked` and `n/a` items MUST equal the total number of rows in the Requirements Matrix. `GAP_COUNT` records the total number of open gap records in the Known Gaps section (i.e. gaps with status `Open`). **Refresh semantics**: these fields MUST be recomputed and rewritten whenever the Requirements Matrix or Known Gaps section is modified — they are never manually edited in isolation.

§4.1.3 Recovery Instructions Section

The Recovery Instructions section MUST appear as the first substantive content after the header fields. It is written for a future Claude session that has no memory of the current conversation and must be actionable without any additional context.

Recovery instructions MUST include:

1. A statement that the reader is implementing a specification (not writing code freely)

2. Instructions to read the full tracker before taking any action
3. Worktree validation steps — check the `Worktree` field and verify the path exists
4. Spec type handling — single-file reads the full spec; multi-file reads the table of contents and rebuilds the structural index
5. Instructions to check the TaskList for pending tasks
6. Instructions to re-read relevant spec sections before any implementation work
7. Sub-agent delegation guidance, including model tier selection
8. A reminder that tests must be run after each sub-agent completes
9. A reminder that the tracker must be updated after each completed task

The recovery workflow is summarised as:

> **Tracker → Spec sections → Sub-agent → Run tests → Verify → Update tracker**

§4.1.4 Specification Summary Section

A brief prose description of what the spec covers and a bulleted list of the major functional areas. This section allows a recovering session to orient quickly without re-reading the full specification.

§4.1.5 Requirements Matrix

The core tracking table. MUST be kept current after every task completion. Format:

```
| Section | Requirement | Priority | Status | Implementation | Tests |
|---------|-------------|----------|--------|----------------|-------|
| §2.1    | In-flight triggers | Must | complete | src/triggers.py:45 | test_triggers.py:12 |
| §2.4    | Merge detection    | Must | partial  | EdgeCaseHandler    | -                   |
| §9.1    | Follow-up extraction | Should | pending | - | - |
```

Status values (requirement-level):

Value	Meaning
`pending`	Not started
`in_progress`	Currently being implemented
`partial`	Partially implemented; gaps identified
`complete`	Fully implemented and verified
`blocked`	Cannot proceed; see Known Gaps
`n/a`	Not applicable to this implementation

Priority values (MoSCoW):

Value	Meaning
`Must`	Required for spec compliance
`Should`	Expected but not blocking
`Could`	Nice to have
`Won't`	Explicitly out of scope

MoSCoW serialisation convention: The tracker uses Title Case (`Must`, `Should`, `Could`, `Won't`) for human readability. Verification fragments use UPPER CASE (`MUST`, `SHOULD`, `COULD`, `WONT`) as enum values in JSON. The canonical mapping is:

Tracker (Title Case)	Fragment JSON (UPPER CASE)
`Must`	`MUST`
`Should`	`SHOULD`
`Could`	`COULD`
`Won't`	`WONT`

The `Implementation` column MUST contain file and line references (`src/file.py:45`) once implementation work is complete. The `Tests` column MUST be populated before a requirement is marked `complete` — a requirement without passing tests is not considered done.

§4.1.6 Structural Index (Multi-File Specs Only)

For multi-file specifications, the tracker MUST include a structural index recording the byte size, estimated token count, and model routing recommendation for each section file. This index serves as the spec baseline — it is used on session recovery to detect spec evolution (new files, removed files, significant size changes, or sub-split patterns).

The index MUST exist in two forms: a machine-readable HTML comment block and a human-readable markdown table.

```
<!-- STRUCTURAL_INDEX
file: sections/01-overview.md | bytes: 3200 | tokens: 800 | route: sonnet | parent: §1
file: sections/04-auth.md | bytes: 88000 | tokens: 22000 | route: opus | parent: §4
-->

| File | Bytes | Est. Tokens | Model Route | Parent Section |
|------|-------|-------------|-------------|----------------|
| `sections/01-overview.md` | 3,200 | 800 | sonnet | §1 |
| `sections/04-auth.md` | 88,000 | 22,000 | opus | §4 |
```

Estimated tokens are computed as `bytes / 4`. Model routing thresholds: < 5,000 tokens → sonnet (group 2–3 together); 5,000–20,000 tokens → sonnet (one per agent); > 20,000 tokens → opus (one per agent).

§4.1.7 Known Gaps Section

Named gap records for requirements that have been discovered to be incomplete or blocked. Each gap MUST include:

- A sequential identifier (`GAP-001`, `GAP-002`, ...)
- Discovery date
- Severity (`High`, `Medium`, `Low`)
- Description
- The specific spec requirement reference
- Current observed behaviour
- Proposed fix
- Status (`Open` / `Resolved`)

§4.1.8 Deviations from Spec Section

Named deviation records for intentional departures from the specification. Each deviation MUST be approved by the developer and MUST include: the spec requirement, the actual implementation, the rationale, the approver, and the date of approval.

§4.1.9 Implementation Log Section

A chronological, session-by-session log of implementation work. Each entry MUST note the date, what was completed, any gaps discovered, and the test status at session end. This log is the primary audit trail for understanding what was done and in what order.

Structured entry format:

```
### 2026-02-19

- **Completed**: §2.1 In-flight triggers (src/triggers.py), §2.4 Merge detection (src/merge.py)
- **Gaps discovered**: GAP-001 — webhook retry logic unspecified in spec
- **Test status**: 47/50 passing; 3 skipped (awaiting GAP-001 resolution)
- **Tracker updated**: Yes
```

Each entry MUST be a level-3 heading containing the ISO date, followed by a bulleted list with at least the `Completed`, `Gaps discovered`, and `Test status` fields.

---

§4.2 Verification Fragments

File pattern: `.impl-verification/<spec-name>/fragments/<fragment-id>.json` **Companion**: `.impl-verification/<spec-name>/fragments/<f` **Created by**: Verification sub-agents (one fragment per requirement) **Read by**: `verify_report.py` during report assembly **Committed to git**: Recommended to be gitignored (see §4.8)

A verification fragment is a single JSON file capturing the verification result for one requirement. One sub-agent is spawned per requirement, and each writes one fragment file to disk upon completion.

§4.2.1 Fragment Schema

The fragment schema is defined in `tools/verification_schema.py`. All fields below are required unless marked optional.

```
{
  "schema_version": "1.0.0",
  "fragment_id": "s02-1-in-flight-triggers",
  "section_ref": "§2.1",
  "title": "In-flight triggers",
  "requirement_text": "The system MUST detect webhook events for in-flight PRs...",
  "moscow": "MUST",
  "status": "implemented",
  "implementation": {
    "files": [
      {
        "path": "src/triggers.py",
        "lines": "45-78",
        "description": "InFlightTrigger class"
      }
```

```
        ],
        "notes": ""
      },
      "test_coverage": "full",
      "tests": [
        {
          "path": "tests/test_triggers.py",
          "lines": "12-34",
          "description": "test_in_flight_trigger_detected"
        }
      ],
      "missing_tests": [],
      "missing_implementation": [],
      "notes": "",
      "v_item_id": "",
      "previous_status": null,
      "resolution": null
    }
```


Field descriptions:

Field	Type	Description
`schema_version`	string	Schema version; currently `"1.0.0"`
`fragment_id`	string	Unique identifier; MUST match the filename stem (e.g. `s02-1-in-flight-triggers` for file `s02-1-in-flight-triggers.json`). Section-number prefixes MUST use zero-padded numerics (e.g. `s01-`, `s02-`, `s10-`) so that lexicographic sort matches numeric order
`section_ref`	string	Specification section reference (e.g. `§2.1`). Best with structured §N.M format; degrades gracefully to prose labels
`title`	string	Short human-readable requirement title
`requirement_text`	string	The verbatim or paraphrased requirement from the specification
`moscow`	enum	MoSCoW priority: `"MUST"`, `"SHOULD"`, `"COULD"`, `"WONT"`
`status`	enum	Verification status: `"implemented"`, `"partial"`, `"not_implemented"`, `"na"`. Verification sub-agents MUST include this field with one of these four values; this is enforced by prompt templates and the validation schema
`implementation.fil`	array of FileRef	Files implementing this requirement, with line ranges
`implementation.notes`	string	Free-form notes about the implementation
`test_coverage`	enum	Test coverage level: `"full"`, `"partial"`, `"none"`
`tests`	array of FileRef	Test files covering this requirement, with line ranges
`missing_tests`	array of string	Descriptions of test scenarios not yet covered
`missing_implementat`	array of string	Descriptions of implementation gaps
`notes`	string	Free-form verification notes
`v_item_id`	string	V-item identifier assigned during report assembly (e.g. `"V1"`); empty string (`""`) in raw fragments. Uses empty string, not null — this field is always a string type

Field	Type	Description
`previous_status`	enum or null	Verification status from the previous verification run (re-verification only); null for initial runs. Note: this uses the *fragment* status enum (`implemented` , `partial` , `not_implemented` , `na`) — it records what the previous *verification* found, not the tracker requirement status
`resolution`	enum or null	Resolution of a prior V-item: `"fixed"` , `"partially_fixed"` , `"not_fixed"` , `"regressed"` ; null for initial runs

FileRef sub-object:

Field	Type	Description
`path`	string	Relative file path from the project root
`lines`	string	Line range (e.g. `"45-78"`) or empty string
`description`	string	Human-readable description of what this file/range implements

§4.2.2 Validation Rules

The `validate_fragment()` function in `verification_schema.py` enforces these hard constraints:

- All required fields must be present
- `implementation` must contain a `files` array
- `moscow` , `status` , and `test_coverage` must be valid enum values
- `fragment_id` must match the filename stem exactly
- `previous_status` and `resolution` (if present and non-null) must be valid enum values

Consistency warnings (non-fatal) are raised for:

- `status: "implemented"` with non-empty `missing_implementation`
- `status: "not_implemented"` with non-empty `implementation.files`
- `test_coverage: "full"` with non-empty `missing_tests`
- `test_coverage: "none"` with non-empty `tests`

§4.2.3 The `.done` Companion File

Each fragment MUST have a companion `.done` marker file with the same stem:

```
.impl-verification/<spec-name>/fragments/s02-1-in-flight-triggers.json
.impl-verification/<spec-name>/fragments/s02-1-in-flight-triggers.done
```

The `.done` file contents are simply the word `done` . The orchestrator's `wait_for_done.py` tool polls for these markers to detect sub-agent completion without consuming the orchestrator's context window on polling loops.

---

§4.3 Verification Reports

JSON: `.impl-verification/<spec-name>/verify-<date>.json` **Markdown**: `.impl-verification/<spec-name>/verify-<date>.md` **Created by**: `verify_report.py` (assembles from fragments after all `.done` markers appear) **Read by**: Orchestrator

for presenting results; passed as `--previous` for re-verification **Committed to git**: Recommended to be gitignored, but has audit trail value (see §4.8)

A verification report is the assembled, authoritative record of a verification run. It is produced deterministically by `verify_report.py` from the fragment files and is the output that the orchestrator presents to the developer.

§4.3.1 JSON Report Structure

The JSON report is the machine-readable primary artifact. Its structure mirrors the `VerificationReport` dataclass in `verification_schema.py`:

```
{
  "schema_version": "1.0.0",
  "report_type": "initial",
  "metadata": {
    "project_name": "billing-system",
    "spec_path": "docs/billing-spec.md",
    "implementation_path": "src/",
    "date": "2026-02-19",
    "run": 1,
    "previous_report": null,
    "spec_version": "",
    "mode": ""
  },
  "findings": [ ... ],
  "statistics": {
    "total_requirements": 24,
    "by_status": { "implemented": 18, "partial": 4, "not_implemented": 2 },
    "by_moscow": { ... },
    "test_coverage": { "full": 15, "partial": 5, "none": 4 },
    "implementation_rate": 0.792,
    "test_rate": 0.729,
    "must_implementation_rate": 0.867
  },
  "priority_gaps": [ ... ],
  "resolution_summary": null
}
```

`resolution_summary` structure (present only in re-verification reports, null otherwise):

```
{
  "resolution_summary": {
    "total_resolved": 5,
    "by_status": {
      "fixed": 3,
      "partially_fixed": 1,
      "not_fixed": 1,
      "regressed": 0
    },
    "unresolved_items": ["V4", "V12"]
  }
}
```

`report_type` is `"initial"` for the first run and `"reverify_delta"` for subsequent runs. When `--previous` is supplied to `verify_report.py`, the report enters re-verification mode: V-item IDs are carried forward from the previous run, and a `resolution_summary` is added showing how many items were fixed, partially fixed, not fixed, or regressed.

Terminology note — `report_type` vs `mode`: These two fields encode the same concept using different vocabularies. `report_type` uses a compact machine-readable enum (`"reverify_delta"`) intended for programmatic comparison and branching. `mode` uses a human-readable label (`"re-verification"`) intended for display in the markdown report header. They are always consistent: when `report_type == "reverify_delta"`, `mode == "re-verification"`; when `report_type == "initial"`, `mode == "initial"`.

Metadata fields:

- `spec_version`: Captures the spec version (date or identifier) at the time of this verification run, for audit traceability. Empty string if unknown.
- `mode`: `"initial"` for the first verification run, or `"re-verification"` for subsequent runs against a previous report. Empty string if unset. See terminology note above for the relationship to `report_type`.

§4.3.2 Markdown Report

The markdown report is rendered by `render_markdown()` in `verification_schema.py`. It is intended for human reading and contains:

- Report header (spec path, date, run number, previous report link)
- Overall summary (implementation rate, test coverage rate)
- Requirement-by-requirement verification listing (each V-item with status, implementation files, test files)
- Previous V-item resolution table (re-verification only)
- Test coverage summary table
- Priority gaps list (high / medium / low)
- Scorecard table (implemented count, test coverage counts, critical gaps)
- Still-open items (re-verification only)
- Recommendations

§4.3.3 V-Item Identifiers

V-item IDs (`V1`, `V2`, ...) are assigned by `assign_v_items()` during initial report assembly. Findings are sorted by `fragment_id` (lexicographic) for deterministic ordering — this is why zero-padded section prefixes (§4.2.1) are required, so that `s02-*` sorts before `s10-*`. On re-verification, `map_v_items_from_previous()` carries forward V-item IDs from the previous run by matching on `section_ref`, and assigns new sequential IDs to any requirements that were not present in the prior run. This ensures that V-item IDs remain stable across runs, making the audit trail traceable.

§4.3.4 Priority Gap Classification

`classify_priority_gaps()` identifies requirements that are not fully implemented with full test coverage and classifies them by priority. This function operates on **fragment status values** (`implemented`, `partial`, `not_implemented`, `na`) — not on tracker requirement status values (`complete`, `in_progress`, etc.). The fragment status reflects what the verification sub-agent observed at verification time.

Priority	Condition
High	MUST + `not_implemented`, or MUST + `partial` + no test coverage, or MUST + `blocked`

Priority	Condition
Medium	MUST + `partial` (with some tests), or MUST + `implemented` (test gap only), or SHOULD + `not_implemented`
Low	SHOULD + `partial`, or COULD + any gap

NA-status items are excluded from gap analysis. `blocked` requirements at MUST priority are classified as High because they represent unresolvable compliance risks.

> **Note**: `blocked` is not a native fragment status value — it is a tracker-level status. When a tracker requirement is `blocked`, its corresponding verification fragment will typically carry `not_implemented` as the fragment status (since the requirement was never successfully implemented). The High-priority classification for MUST + `blocked` is applied by the report assembler based on the tracker state at the time of verification, not from the fragment alone.

---

§4.4 Implementation Work Directory

Location: `.impl-work/<spec-name>/` **Created by**: Orchestrator before dispatching sub-agents **Written by**: Implementation sub-agents (structured JSON output) **Read by**: Orchestrator after sub-agents complete **Committed to git**: MUST be gitignored (build artifacts)

The implementation work directory is a scratch space for structured output from sub-agents during Phase 2 (Implementation) and fix cycles. Sub-agents write structured JSON files here rather than returning conversational responses, keeping the orchestrator's context window clean.

§4.4.1 Files in the Work Directory

File	Written by	Contents
`summary.json`	Implementation sub-agent	Status (`done` / `partial` / `failed`), concerns, digest, list of files changed
`compliance.json`	Spec compliance check agent	Compliance findings, issues, recommendations
`fix-summary.json`	Fix agent	What was fixed, what remains, test results

Each of these files MUST have a companion `.done` marker (e.g. `summary.done`) that signals completion to the orchestrator's polling loop.

> **Note**: `compliance.json` and `fix-summary.json` are intentionally loosely structured. They are written by sub-agents for debugging and documentation purposes and are not validated by the orchestrator or used in control flow decisions. Their minimal expected shapes are documented below, but additional fields may be present.

`compliance.json` expected shape:

```
{
  "issues": [ { "description": "...", "severity": "..." } ],
  "recommendations": [ "..." ],
  "overall": "pass | partial | fail"
}
```

`fix-summary.json` **expected shape**:

```
{
  "fixed": [ "..." ],
  "remaining": [ "..." ],
  "test_results": "pass | partial | fail"
}
```

§4.4.2 summary.json Structure

```
{
  "status": "done",
  "concerns": [],
  "digest": "Implemented InFlightTrigger class in src/triggers.py. Added webhook...",
  "files_changed": [
    "src/triggers.py",
    "tests/test_triggers.py"
  ]
}
```

`status` values: `"done"` (fully complete), `"partial"` (incomplete with concerns noted), `"failed"` (could not complete).

Advisory mapping to tracker requirement status: `done` maps to `complete`, `partial` maps to `partial`, `failed` maps to `in_progress` (for tracker update purposes, since a failed attempt means the requirement is still being worked on). This mapping is advisory — the orchestrator and user interpret these values when updating the tracker, but it is not enforced programmatically.

`digest` is a concise summary of what was done, used by the orchestrator to decide whether to escalate to a higher-tier model review. Digest signals for mandatory escalation include: algorithms, state machines, permission/auth logic, complex business rules, and cross-cutting concerns. The full list of escalation signal keywords and the detection algorithm are specified in §5.4 (Digest-Based Escalation).

§4.4.3 .done Markers in the Work Directory

`.done` markers MUST be cleared by the orchestrator before each sub-agent dispatch batch. This prevents stale markers from prior runs from falsely signalling completion. The clearing step is mandatory and is enforced by the workflow as a precondition.

---

§4.5 Preferences Files

Project scope: `.impl-preferences.md` (at project root) **Global scope**: `~/.claude/.impl-preferences.md` **Created by**: Developer (manually) or `/implement config` command **Read by**: Phase 1 (Planning), when determining default TDD mode and other settings **Committed to git**: Project-scope file SHOULD be committed; global file is user-local

Preferences files are simple markdown documents with key-value pairs that configure default skill behaviour. When both files exist, the project-scope file takes precedence over the global file for any keys it defines.

§4.5.1 Format

```
# Implementation Preferences

tdd_mode: on
default_model: sonnet
worktree_prefix: feature/
```

All keys are optional. The skill reads only the keys it recognises and ignores unknown keys. Preferences are applied during Phase 1 planning and are recorded into the tracker header fields, where they take effect for the rest of the implementation lifecycle.

§4.5.2 Recognised Keys

Key	Values	Default	Description
`tdd_mode`	`on` / `off` / `ask`	`on`	Whether to activate TDD workflow by default
`default_model`	`haiku` / `sonnet` / `opus`	`sonnet`	Default model tier for implementation sub-agents. See interaction note below.
`worktree_prefix`	string	*(none)*	Prefix to prepend to auto-generated worktree branch names

`default_model` and algorithmic routing: The structural index (§4.1.6) records a per-section model routing recommendation derived from estimated token counts (< 5,000 tokens → sonnet; > 20,000 tokens → opus). `default_model` acts as a **user-supplied floor**: when set, no sub-agent is dispatched to a tier lower than `default_model`, but the algorithmic routing may still upgrade to a higher tier (e.g. if `default_model: sonnet` but a section exceeds 20,000 tokens, the section is still routed to opus). Setting `default_model: opus` effectively disables downgrade routing and sends all sub-agents to opus regardless of token count.

---

§4.6 Completion Markers

Pattern: `<path>/<name>.done` **Created by**: Sub-agents (upon task completion) **Read by**: `wait_for_done.py` (polling tool) **Contents**: The single word `done`

Completion markers are sentinel files that signal a sub-agent has finished its task and written its output to disk. The orchestrator uses `wait_for_done.py` to block on these markers rather than polling with repeated `ls` calls, which would waste context window on tool call noise.

§4.6.1 wait_for_done.py Modes

The tool supports two invocation modes:

```
# Wait for N .done files in a directory
python wait_for_done.py --dir .impl-work/billing-spec/ --count 3

# Wait for specific named files
python wait_for_done.py \
  --files .impl-work/billing-spec/summary.done \
          .impl-work/billing-spec/compliance.done
```

Default timeout is 600 seconds (10 minutes). Exit code 0 on success, 1 on timeout.

§4.6.2 Clearing Markers

Before dispatching a new batch of sub-agents, the orchestrator MUST delete any `.done` files from the previous batch for the same directory. This is a hard precondition — failure to clear markers will cause `wait_for_done.py` to return immediately with stale results.

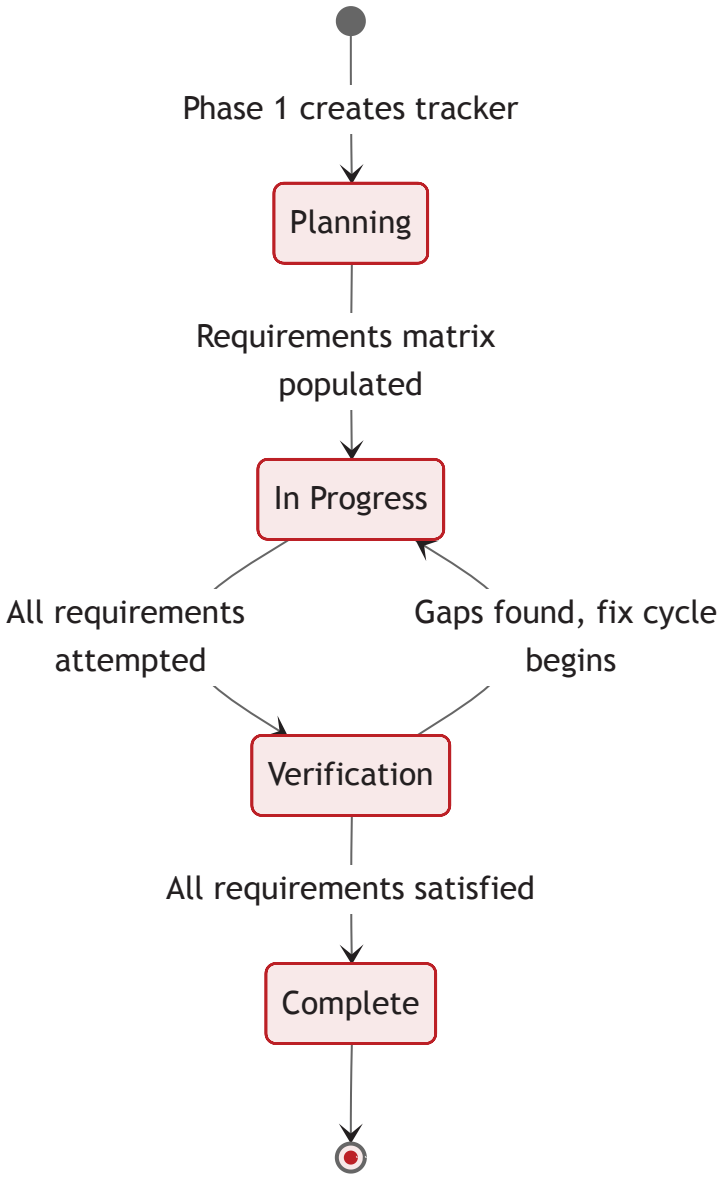
---

§4.7 State Machines and Lifecycle

> **Advisory documentation**: The state machines in this section document the *intended* lifecycle of tracker, requirement, and V-item statuses. They are not programmatically enforced — the skill uses simple status tracking, and the orchestrator does not validate transitions against a formal state machine. These diagrams serve as reference documentation for implementors and reviewers.

§4.7.1 Tracker Status Transitions

The overall tracker `Status` field follows a linear progression through four states.



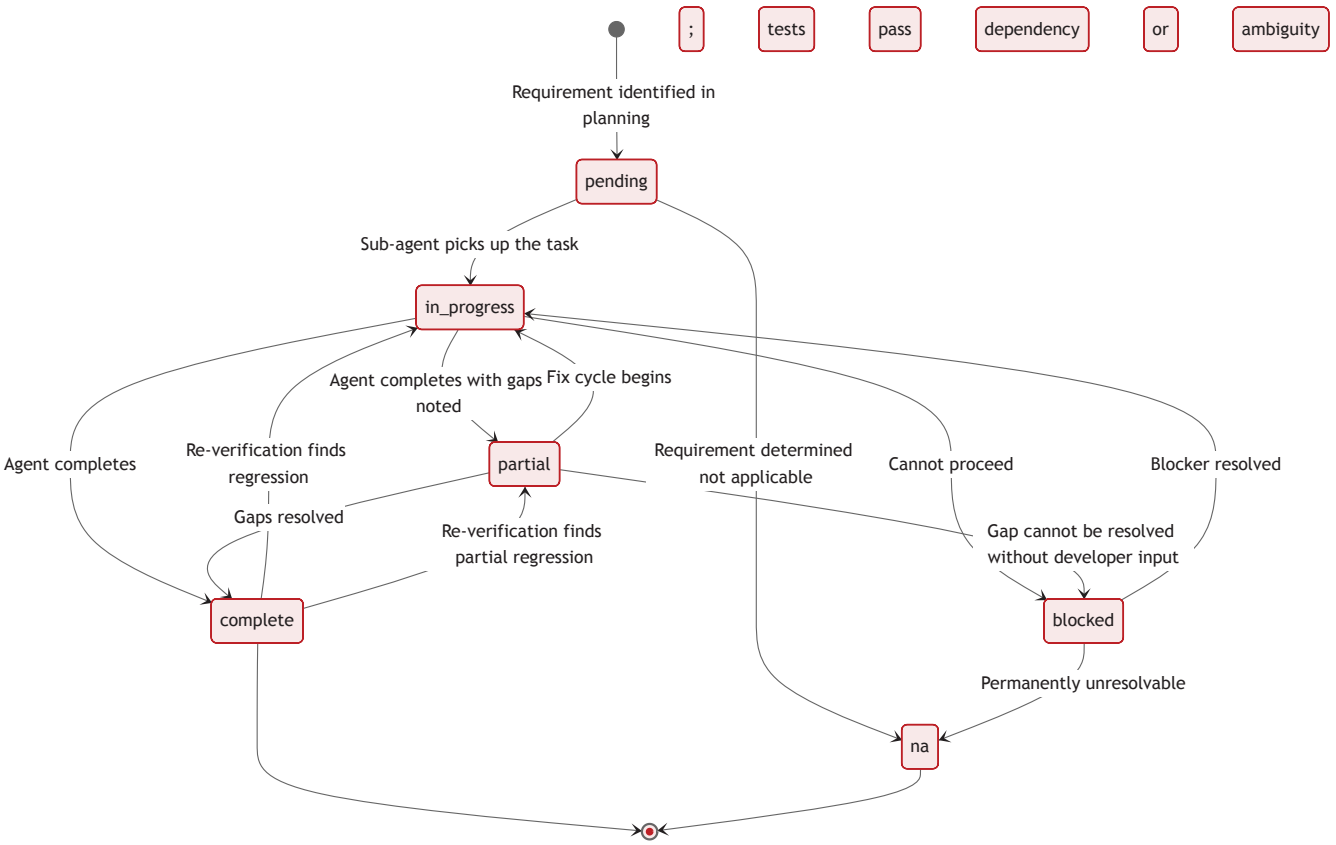
Transitions (advisory):

From	To	Trigger
(new)	Planning	Phase 1 creates the tracker
Planning	In Progress	Implementation sub-agents begin first task. **Precondition**: the requirements matrix should be populated before implementation begins
In Progress	Verification	All requirements in the matrix have been attempted (see definition of "attempted" below)
Verification	In Progress	Verification report contains unresolved gaps; fix cycle begins
Verification	Complete	All requirements are `complete` or `n/a`; no high-priority gaps remain

Definition of "attempted": A requirement is considered "attempted" when it is in any status other than `pending` — i.e., `in_progress`, `partial`, `complete`, `blocked`, or `n/a`.

§4.7.2 Requirement Status Transitions

Individual requirement rows in the Requirements Matrix transition through their own lifecycle.



Status value semantics:

Status	Meaning	Next Step
`pending`	Not yet started	Assign to next sub-agent batch
`in_progress`	Active in current session	Await sub-agent completion
`partial`	Implemented with identified gaps	Schedule fix cycle
`complete`	Implemented and tests passing	No action required (may regress on re-verification)
`blocked`	Cannot proceed	Escalate to developer; may transition to `n/a` if permanently unresolvable
`n/a`	Determined not applicable	Document rationale in tracker

Regression transitions (advisory): When re-verification discovers that a previously `complete` requirement has regressed, it may be moved back to `in_progress` (triggered by a `regressed` resolution finding) or to `partial` (triggered by a `partially_fixed` finding on re-check). These transitions are advisory — the user updates the tracker based on the verification report.

§4.7.3 V-Item Resolution Values (Re-Verification)

When a requirement appeared in a previous verification run and is being re-checked, the `resolution` field records the outcome of the intervening fix cycle:

Value	Meaning
`fixed`	Was non-implemented or partial; now fully implemented with full test coverage
`partially_fixed`	Some progress made but requirement still not fully satisfied
`not_fixed`	No change from previous run
`regressed`	Was implemented (or partial) in previous run; now in a worse state

§4.7.4 Cross-Machine Synchronisation

> **Advisory**: The mappings below document how status values in different artifacts relate to each other. The skill does not enforce these mappings programmatically — the user reads verification reports and updates the tracker manually.

(a) Fragment status to requirement status:

Fragment `status`	Tracker requirement status
`implemented`	`complete`
`partial`	`partial`
`not_implemented`	`in_progress` if a sub-agent attempted implementation but produced no usable output; `pending` if the requirement was never dispatched to a sub-agent at all
`na`	`n/a`

The distinction for `not_implemented`: check the Implementation Log (§4.1.9) and the `implementation.files` array in the fragment. If `implementation.files` is non-empty or the log shows a sub-agent was dispatched for this requirement, use `in_progress`. If neither is true, the requirement was never attempted and should remain `pending`.

(b) summary.json status to requirement status:

`summary.json` `status`	Tracker requirement status
`done`	`complete`
`partial`	`partial`
`failed`	`in_progress`

(c) V-item resolution to requirement status:

V-item `resolution`	Tracker requirement status
`fixed`	`complete`
`partially_fixed`	`partial`

V-item `resolution`	Tracker requirement status
`not_fixed`	*(unchanged)*
`regressed`	`in_progress`

(d) Aggregate conditions for tracker-level transitions:

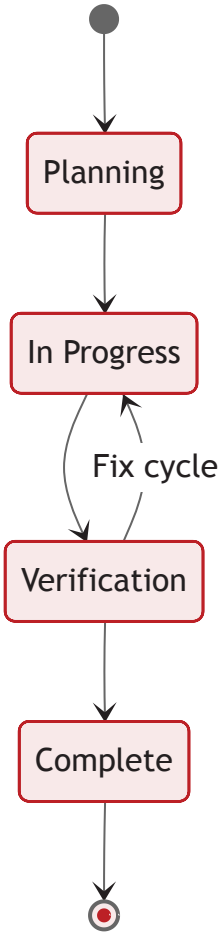
Condition	Tracker status transition
All requirements in a status other than `pending`	In Progress to Verification
All requirements `complete` or `n/a`, no high-priority gap	Verification to Complete
Any requirement not `complete` or `n/a` after verification	Verification to In Progress (fix cycle)

§4.7.5 State Machine Reference

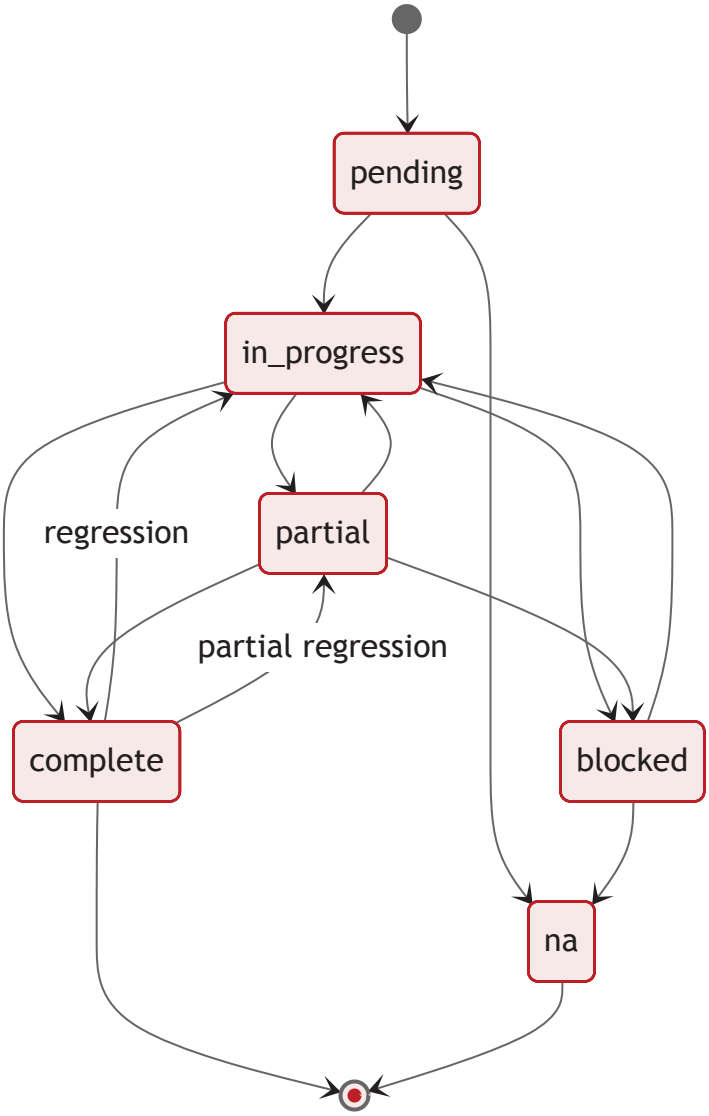
This subsection collects all state machine diagrams in one place for quick reference.

> **Important**: These are advisory documentation of the intended lifecycle, not enforced contracts. The skill uses simple status tracking and does not validate transitions programmatically.

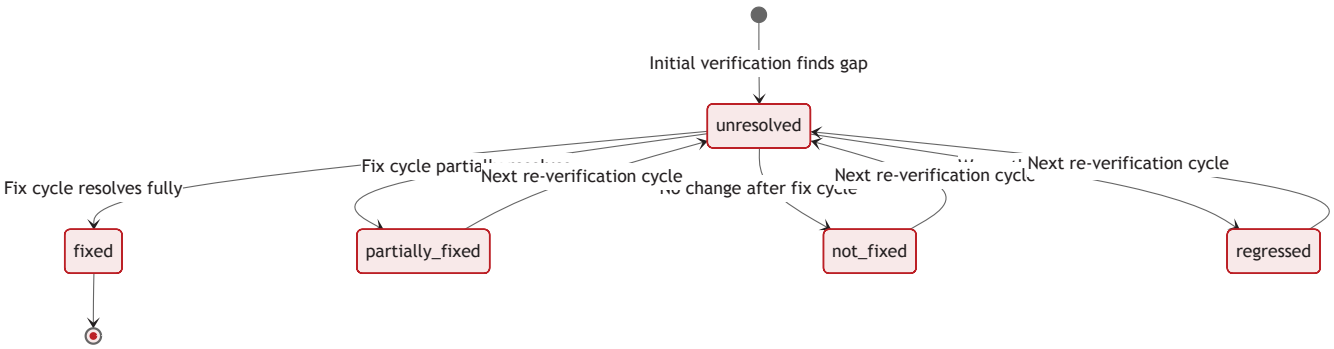
Tracker lifecycle:



Requirement lifecycle:



V-item resolution lifecycle (re-verification only):



§4.8 File Lifecycle & Gitignore

§4.8.1 What to Commit

Artifact	Commit?	Rationale
`.impl-tracker-<spec-name>.md`	MUST	Persistent bridge across sessions; loss of this file means loss of all implementation state
`.impl-preferences.md` (project)	SHOULD	Encodes team-level defaults; useful for collaborators
`.impl-verification/<spec-name>/verify-<date>.json`	Optional	Has audit trail value; larger projects may prefer to gitignore
`.impl-verification/<spec-name>/verify-<date>`	Optional	Human-readable audit trail; same rationale as JSON
`.impl-verification/<spec-name>/fragments/`	SHOULD NOT	Build artifacts; regenerated on each verification run
`.impl-work/<spec-name>/`	MUST NOT	Transient scratch space; regenerated on each implementation batch

§4.8.2 Recommended .gitignore Entries

```
# Implementation skill — transient build artifacts
.impl-work/
.impl-verification/
```

This is a recommendation, not a hard requirement. The developer MAY choose to commit verification reports if they want a persistent audit trail of verification history. This can be valuable for compliance-sensitive projects or for tracking specification fidelity over time.

Tracker files (`.impl-tracker-*.md`) are explicitly excluded from the gitignore recommendation. They MUST be committed because they are the only durable link between the specification and the ongoing implementation work.

§4.8.3 Directory Layout Summary

A complete project using the skill will have the following artifact layout:

```
<project-root>/
├── .impl-tracker-my-spec.md          # COMMIT: persistent tracker
├── .impl-preferences.md               # COMMIT: project-level preferences
│
├── .impl-work/                        # GITIGNORE: transient sub-agent output
│   └── my-spec/
│       ├── summary.json
│       ├── summary.done
│       ├── compliance.json
│       └── compliance.done
│
└── .impl-verification/                # OPTIONAL: verification reports + fragments
    └── my-spec/
        ├── verify-2026-02-19.json     # Assembled report (JSON)
        ├── verify-2026-02-19.md       # Assembled report (Markdown)
        └── fragments/
```

```
        ├── s02-1-in-flight.json   # Per-requirement fragment
        ├── s02-1-in-flight.done   # Completion marker
        ├── s02-2-closure.json
        └── s02-2-closure.done
```

When a git worktree is active (see §6), each worktree maintains its own `.impl-tracker-*.md` at its own root. The `.impl-work/` and `.impl-verification/` directories may coexist in the main worktree root or within each worktree depending on how the orchestrator is invoked.

§5 Sub-Agent Orchestration

Sub-agent orchestration is the execution engine of the `/implement` skill. The orchestrator — the main Claude Code conversation — never implements directly. It reads trackers, dispatches focused tasks to fresh sub-agents, interprets their structured output, and updates state. This division is not an implementation convenience; it is the primary mechanism by which specification drift is made structurally impossible.

This section specifies the orchestrator-agent architecture, the context isolation model that makes it work, the model selection strategy, the DIGEST-based escalation protocol, the output-to-disk pattern, parallel dispatch coordination, prompt templates, and the boundaries between orchestrator and sub-agent responsibilities.

---

§5.1 Orchestrator-Agent Architecture

The skill operates as a hub-and-spoke system. The orchestrator occupies the hub: it holds the planning state, reads tracker files, makes delegation decisions, and communicates with the developer. Sub-agents occupy the spokes: each receives a focused task brief, performs a single unit of work, writes structured output to disk, and terminates.

§5.1.1 Orchestrator Responsibilities

The orchestrator MUST:

- Read the implementation tracker before each delegation decision
- Re-read the relevant spec sections to prepare sub-agent task briefs
- Select the appropriate model tier for each task (see §5.3)
- Dispatch sub-agents with focused prompts containing only the information needed for that task
- Read sub-agent output from disk (never from conversational response)
- Check DIGEST fields against the complexity escalation table (see §5.4)
- Run tests after each sub-agent completes
- Update the tracker after validating sub-agent output
- Present status summaries and decisions to the developer

The orchestrator MUST NOT:

- Implement code directly
- Write tests directly
- Perform verification judgments in the main conversation
- Read full spec documents into the main context when delegating (pass section references or file paths instead)

§5.1.2 Sub-Agent Responsibilities

Each sub-agent MUST:

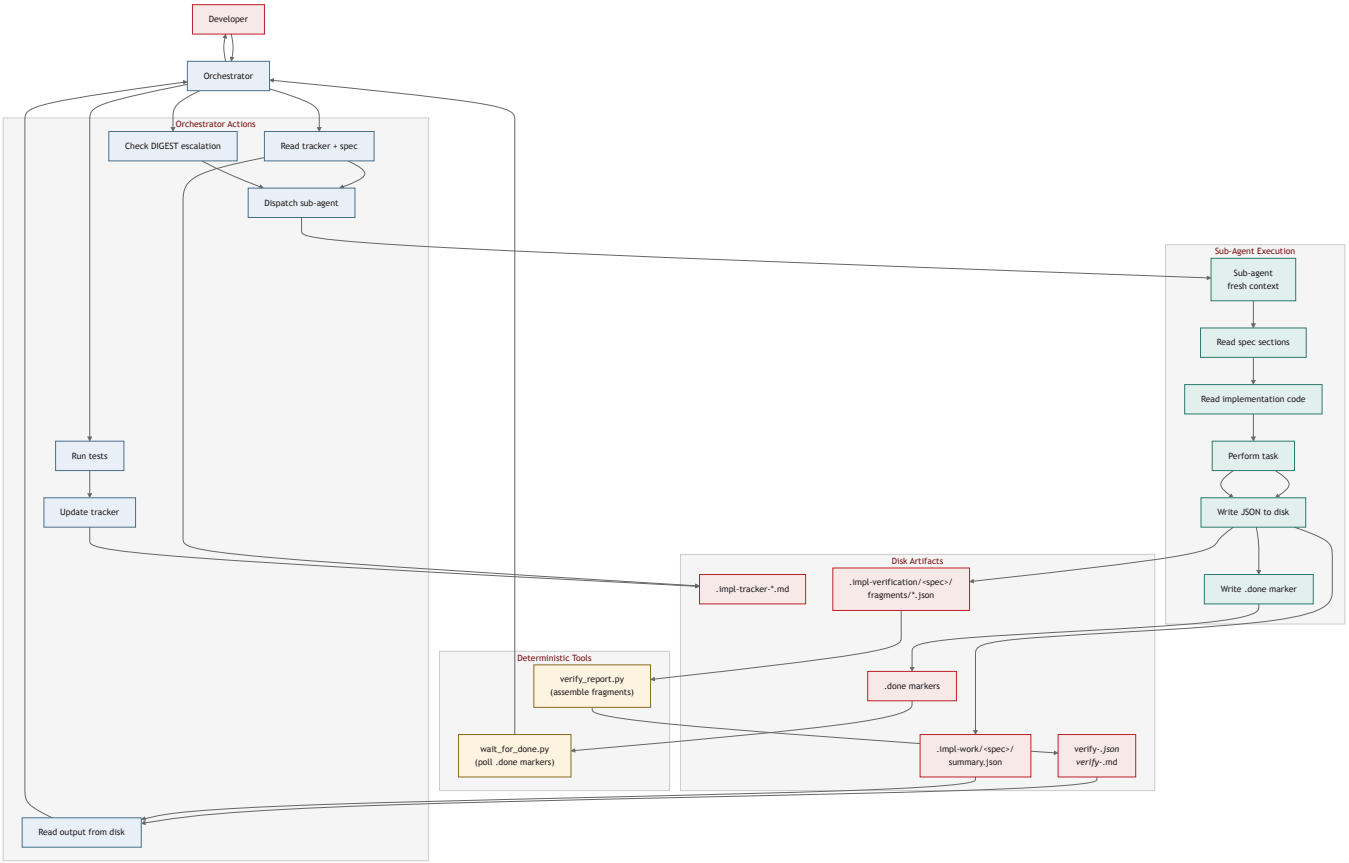
- Read the spec sections and code files specified in its task brief
- Perform the assigned task (implement, test, verify, or fix)
- Write structured JSON output to its designated disk path
- Write a `.done` marker as the final file operation
- Respond with a minimal conversational acknowledgment (e.g., "Done.")

Each sub-agent MUST NOT:

- Read or modify the implementation tracker
- Make decisions about task ordering or priority
- Communicate with other sub-agents
- Return substantive findings in conversational output (all findings go to disk)

§5.1.3 Data Flow

The following diagram shows the complete data flow between the orchestrator, sub-agents, deterministic tools, and disk artifacts.



---

§5.2 Context Isolation

Context isolation is the key insight that makes the entire architecture work. It is not a side-effect of using sub-agents — it is the reason for using them. Every other architectural decision — the output-to-disk pattern, the prompt templates, the model selection strategy — exists to support and exploit context isolation.

§5.2.1 The Core Mechanism

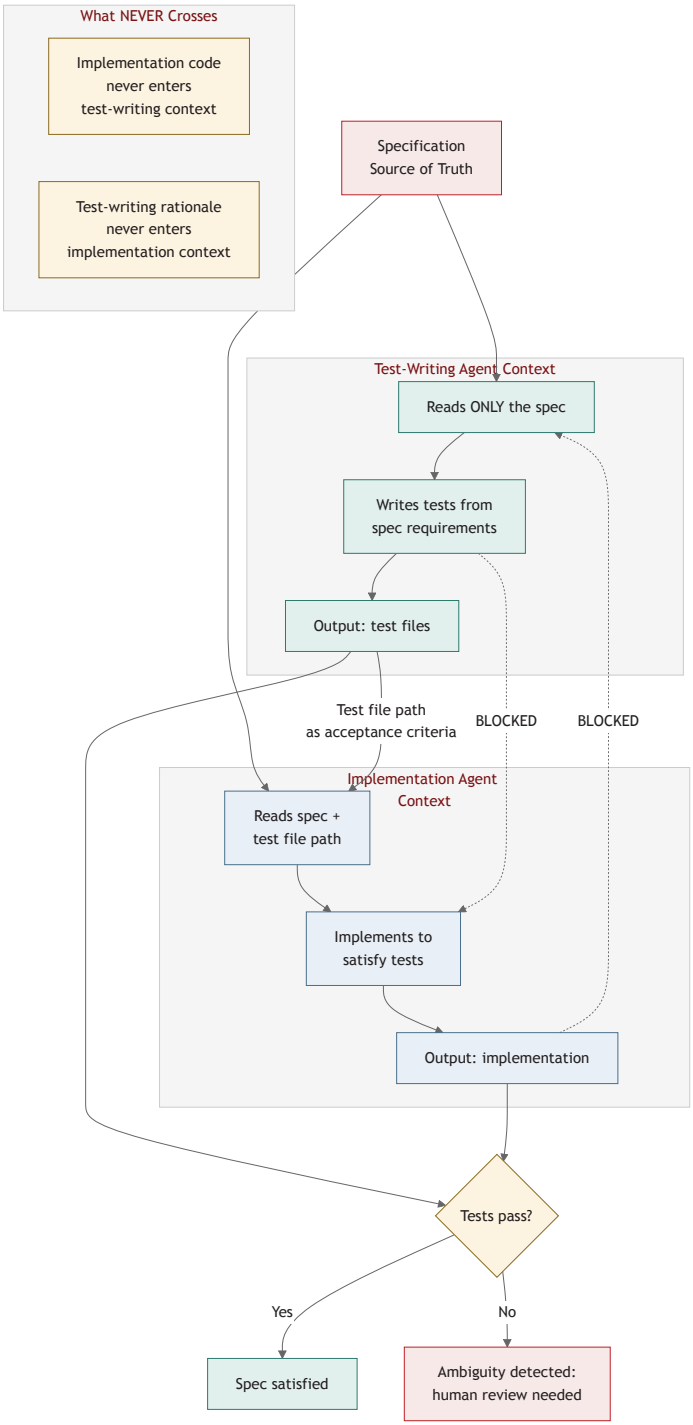
When the orchestrator spawns a sub-agent, that agent starts with a fresh context window. It has no memory of previous implementation turns, no accumulated assumptions, no residual biases from earlier tasks. It reads the specification text and the implementation code as if encountering them for the first time.

This is structurally incapable of drift. The sub-agent cannot have drifted from the specification because it has never seen the specification before this moment. It interprets the spec independently, and any divergence

between its interpretation and the existing code surfaces as a concrete finding — a test failure, a verification gap, a compliance issue.

§5.2.2 Context Isolation in TDD Mode

TDD mode exploits context isolation most aggressively. The test-writing agent and the implementation agent operate in completely separate context windows, each independently interpreting the specification. Their disagreements surface as test failures — which are, in fact, specification ambiguities that require human resolution.



The critical boundaries are:

- The test-writing agent MUST see ONLY the specification. It MUST NOT see any implementation code. This ensures tests verify what the spec requires, not what the code does.
- The implementation agent MUST receive the test file path as acceptance criteria. It sees the spec for context, but the tests define the concrete pass/fail signals.

- When tests fail, the failure is treated as a potential specification ambiguity first, not as an implementation bug first. The developer SHOULD review whether the test or the implementation more faithfully represents the spec's intent.

§5.2.3 Context Isolation in Verification

Each verification sub-agent receives a single requirement and the relevant code paths. It MUST NOT receive the implementing agent's reasoning, assumptions, or summary. The verification agent reads the spec and the code independently, producing a judgment that is structurally independent of the implementer's perspective.

This independence is what makes verification meaningful. A verification agent that inherits the implementer's context would be biased toward confirming the implementation — it would check whether the code matches the implementer's understanding, not whether the code matches the specification.

§5.2.4 Context Isolation in Fix Agents

When fixing verification gaps, the fix agent receives:

- The spec requirement (exact quote)
- The current code (file paths and relevant lines)
- The gap description (from the verification report)

It MUST NOT receive the original implementer's context or the original implementer's summary. The fix agent forms its own understanding of the requirement and the gap, avoiding the risk of inheriting the same misunderstanding that caused the gap.

---

§5.3 Model Selection Strategy

Model selection is per-task, not per-session. Different tasks have different complexity profiles, and the skill MUST match the model tier to the task's reasoning requirements. This is a cost-effectiveness measure: using opus for boilerplate wastes resources; using haiku for verification misses subtle gaps.

§5.3.1 Task-Based Model Selection

Task Type	Model	Rationale
Boilerplate, simple field additions, config changes	`sonnet` (default); `haiku` only for pure template fill-in (e.g., tracker initialization)	Mechanical tasks requiring minimal reasoning. Use haiku only when the task is a pure template fill-in with no context understanding required. When in doubt, default to sonnet.
Standard implementation, moderate complexity	`sonnet`	Good balance of capability and cost for most implementation work
Complex logic, algorithms, state management	`opus`	Requires deep reasoning about correctness
Verification (all)	`opus`	MUST always use opus — catches subtle gaps that smaller models miss
Gap fixing (all)	`opus`	MUST always use opus — requires understanding root cause

Task Type	Model	Rationale
Spec compliance checks	`sonnet` (default), `opus` (complex tasks)	Most checks are straightforward; escalate for complex logic
TDD test writing	`sonnet` (default), `opus` (complex logic)	Most test writing is mechanical; escalate for algorithmic tests
TDD implementatior	`sonnet` (default), `opus` (complex logic)	Same escalation pattern as test writing

The general rule: if a task involves logic decisions, conditional behaviour, or algorithmic thinking, it SHOULD use `opus`. When in doubt, the orchestrator SHOULD prefer `opus` — the cost savings from using smaller models are not worth the risk of missing implementation details.

§5.3.2 Size-Based Routing for Multi-File Specs

When a specification has breakout section files (as produced by `/spec`), the orchestrator MUST build a structural index before delegating work. The index determines model selection and task grouping.

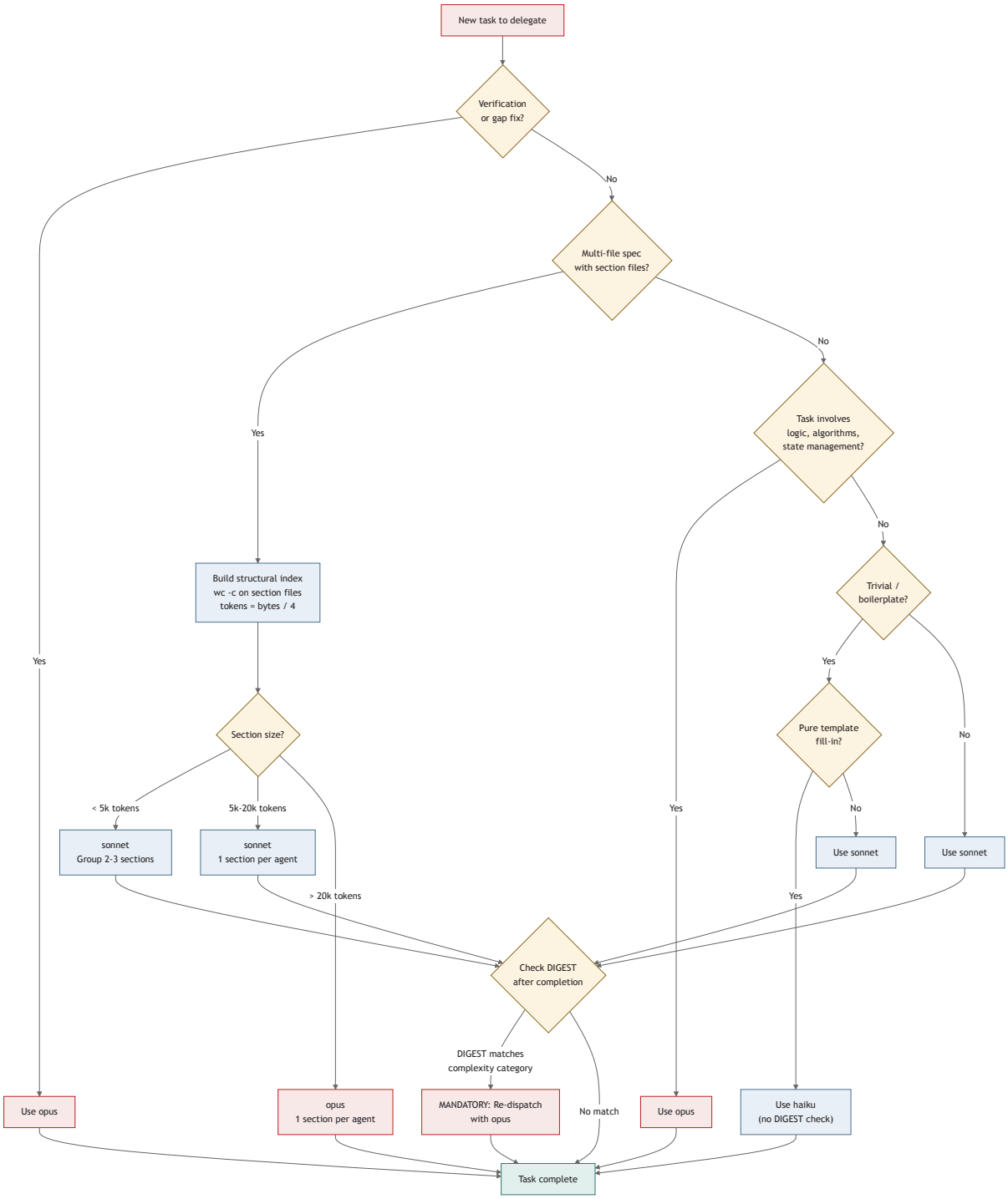
Step 1 — Build the index: Run `wc -c` on each section file. Estimate tokens using: `estimated_tokens = file_size_bytes / 4`. This is an approximation — actual token counts vary by content. Files near the 5k/20k routing boundaries (within approximately 500 tokens) may be misrouted. This is accepted as low-impact: misrouting to a higher tier (more capable model) is safe but slightly more expensive, while misrouting to a lower tier triggers DIGEST escalation (§5.4) as a safety net.

Step 2 — Route by section size:

Section Size	Model	Grouping Strategy
< 5k tokens	`sonnet`	Group 2-3 sections per agent
5k-20k tokens	`sonnet`	1 section per agent
> 20k tokens	`opus`	1 section per agent

Step 3 — Handle sub-split sections: When a section has been split into sub-files (e.g., `02a-`, `02b-`, `02c-`), each sub-file MUST route independently by its own size. Tasks referencing the parent section (e.g., §2) MUST include all sub-file paths in the sub-agent prompt.

§5.3.3 Model Selection Decision Tree



§5.4 DIGEST-Based Escalation

DIGEST-based escalation is the mechanism by which sonnet-class agents signal that their work may require opus-level review. This is not a suggestion system — it is a mandatory escalation protocol. When a DIGEST matches a complexity category, opus re-dispatch MUST occur.

§5.4.1 How DIGEST Works

Every sonnet and opus sub-agent writes a `digest` field in its `summary.json` output. The digest contains three sub-fields:

```
{
  "digest": {
    "entities": "key classes, models, services touched",
    "patterns": "design patterns used or encountered",
    "complexity": "any algorithmic, state machine, auth, or business rule complexity"
  }
}
```

After reading `summary.json` from disk, the orchestrator MUST check the `digest.complexity` sub-field — and only that sub-field — against the complexity category table (§5.4.2). The `entities` and `patterns` sub-fields are informational context for the orchestrator but are NOT inputs to the escalation matching logic.

§5.4.2 Complexity Categories (Canonical Source)

> **This table is the single authoritative source for complexity categories and signal keywords.** Any reference to complexity categories elsewhere in this specification (including §6.5 and §7.2) is a convenience cross-reference and MUST defer to this table in cases of conflict.

Category	DIGEST Signals	Why Opus Review is Required
Algorithms	"algorithm", "calculation", "formula", "heuristic"	Subtle correctness issues in edge cases
State machines	"state machine", "state transition", "lifecycle"	Complex interaction patterns between states
Permission / auth	"permission", "role inheritance", "RBAC", "access control"	Security boundary correctness
Complex business rules	"conditional", "override", "exception", "cascading"	Edge case coverage
Cross-cutting concerns	"affects all", "global constraint", "system-wide"	Holistic view required

Matching mechanism: The orchestrator performs simple substring matching of each signal keyword against the `digest.complexity` text. There is no stemming, fuzzy matching, or semantic analysis — only literal substring containment.

Known limitation — negation false positives: Because matching is substring-based without negation handling, phrases such as "No algorithmic complexity" will trigger the "algorithm" signal. This is accepted as a design trade-off: false-positive escalation (an unnecessary opus review) is low-cost, while false-negative escalation (missing genuine complexity) is high-cost. The system intentionally errs on the side of over-escalation.

§5.4.3 Consolidated Escalation Procedure

The following numbered procedure is the authoritative description of the DIGEST escalation flow. The subsections that follow (§5.4.4, §5.4.5) provide supporting rationale and anti-patterns.

1. After a sonnet agent completes, read `summary.json` from disk. **Scope note**: DIGEST checking applies only to sonnet agents. Haiku agents do not produce a `digest` field and are never subject to escalation. Opus agents are never subject to escalation (see step 6).
2. Extract the `digest.complexity` field (a string). If the `digest` field is absent or malformed (not a JSON object, or `complexity` sub-field is not a string): log a warning and skip escalation (fail-open — do not block progress)
3. Check for substring matches of each signal keyword in the complexity category table (§5.4.2) against the `digest.complexity` text
4. If any signal keyword matches: MUST dispatch an opus review agent using the opus escalation template (§5.7.1). This is mandatory — see §5.4.4 for anti-patterns that must not be used to bypass escalation
5. If no signal keyword matches: proceed to the next task
6. If the agent was pre-dispatched as opus: skip the DIGEST check entirely (opus does not need self-review). This applies to opus agents dispatched via pre-dispatch routing (§5.3) as well as opus escalation review agents themselves.

§5.4.4 Escalation is MANDATORY

When a DIGEST matches any complexity category, the orchestrator MUST dispatch an opus agent to review the sonnet's code changes. This is a hard gate (see §2.2, Principle 2). The escalation is not discretionary, not conditional on the orchestrator's assessment of severity, and not optional when time or context is short.

The opus agent receives the sonnet's work as a starting point and goes deeper on the flagged complexity area. It does not start from scratch — it reviews and improves what the sonnet produced.

§5.4.5 Anti-Patterns: Invalid Reasons to Skip Escalation

The following rationalisations are NOT valid reasons to skip DIGEST escalation. The orchestrator MUST NOT use any of these to bypass the mandatory opus re-dispatch:

Rationalisation	Why It Is Invalid
"The sonnet agent's work looks correct"	The orchestrator is not performing verification — it delegates that. Looking correct is not the same as being correct, especially for the complexity categories listed above.
"The DIGEST match is borderline"	There is no borderline. If the signal word appears in the digest, escalation is triggered. Ambiguity is resolved in favour of escalation, not in favour of skipping.
"We're running low on context / time"	Context pressure is exactly the condition under which the orchestrator is most likely to make poor judgment calls. Escalation exists to compensate for this.
"The task was simple despite the DIGEST signal"	The agent that performed the task flagged complexity. Overriding that assessment from the orchestrator's more distant vantage point is not valid.
"We can catch it during verification"	Verification is a separate phase with a separate purpose. DIGEST escalation catches implementation-time issues that may be harder to diagnose at verification time.

Rationalisation	Why It Is Invalid
"The tests pass"	Passing tests demonstrate that the implementation satisfies the test suite, not that it satisfies the specification. The complexity categories above specifically concern areas where the gap between "tests pass" and "spec satisfied" is widest.

The purpose of documenting these anti-patterns explicitly is that LLMs under context pressure tend toward exactly these rationalisations. By naming them in advance, the skill makes it harder for the orchestrator to convince itself that skipping is acceptable.

§5.4.6 Escalation Workflow (Detailed)

> **Note**: The consolidated escalation procedure (§5.4.3) is the authoritative reference for the matching and dispatch logic. This subsection describes the full end-to-end workflow including the opus agent's responsibilities.

1. Sub-agent (sonnet) completes task and writes `summary.json` with `digest` field
2. Orchestrator reads `summary.json` from disk
3. Orchestrator checks `digest.complexity` against the category table (§5.4.2, canonical) using substring matching (see §5.4.3 steps 2-5)
4. If any signal matches: MUST dispatch opus agent with the sonnet's changed files and the flagged complexity area, using the opus escalation template (§5.7.1)
5. Opus agent reviews and, if needed, improves the sonnet's work
6. Opus agent writes its own `summary.json` (overwriting the sonnet's) and MUST write a `.done` marker, same as all other sub-agents. When writing `summary.json`, the opus agent SHOULD preserve the sonnet's `files_changed` list as a baseline and extend it with any additional file changes made during review. If the opus agent makes no additional file changes, it MUST still include the sonnet's `files_changed` list verbatim so that downstream tools retain the full file-change manifest.
7. Orchestrator reads the updated summary and proceeds with tests

Pre-dispatch vs post-dispatch escalation: Pre-dispatch complexity assessment is the primary routing mechanism — the orchestrator uses judgment plus section size (§5.3.2) to select the initial model tier. Post-dispatch DIGEST escalation is the safety net — it catches complexity that was not apparent before the agent began working. Both mechanisms may fire for the same task; this is intentional (belt and suspenders). If a task was pre-dispatched to opus, the DIGEST check is skipped because opus does not need self-review.

---

§5.5 Output-to-Disk Pattern

All sub-agents write structured output to disk rather than returning findings in conversational responses. This is a hard requirement — the orchestrator MUST read sub-agent results from disk files, never from `TaskOutput`.

§5.5.1 Rationale

Conversational output from sub-agents has three problems:

1. **It consumes orchestrator context.** Reading a sub-agent's conversational response loads that content into the main conversation's context window — exactly the resource the skill is designed to protect.

2. **It is unstructured.** Conversational responses are prose that the orchestrator must parse, introducing opportunities for misinterpretation.
3. **It is ephemeral.** Once the conversation moves on, the sub-agent's response is subject to compaction. Disk artifacts persist indefinitely.

Writing to disk solves all three: the orchestrator reads a compact JSON file (structured, parseable, persistent) instead of a verbose conversational response.

§5.5.2 Output Paths by Agent Type

Agent Type	Output Path	Format
Implementation agent	`.impl-work/<spec-name>/summary.json`	Task summary with files changed, concerns, digest
Compliance check agent	`.impl-work/<spec-name>/compliance.json`	Verdict with issues list
TDD test-writing agent	`.impl-work/<spec-name>/tdd-tests-summary.json`	Test file paths written, spec requirements covered, digest
TDD implementation agent	`.impl-work/<spec-name>/tdd-impl-summary.json`	Files changed, test results summary, digest
Verification agent	`.impl-verification/<spec-name>/fragments/<id>.json`	Per-requirement finding
Fix agent	`.impl-work/<spec-name>/fix-summary-<gap-id>.json`	Fix description with files changed (one file per gap, keyed by gap ID to avoid overwrite when fix agents run concurrently)

Each output file is accompanied by a `.done` marker at a predictable path:

Agent Type	Marker Path
Implementation / compliance agent	`.impl-work/<spec-name>/summary.done` or `compliance.done`
TDD test-writing agent	`.impl-work/<spec-name>/tdd-tests-summary.done`
TDD implementation agent	`.impl-work/<spec-name>/tdd-impl-summary.done`
Verification agent	`.impl-verification/<spec-name>/fragments/<id>.done`
Fix agent	`.impl-work/<spec-name>/fix-summary-<gap-id>.done`

§5.5.3 The .done Marker Protocol

The `.done` marker MUST be the last file a sub-agent writes. Its presence signals that the JSON output file is complete and safe to read. The marker contains only the word "done" — its content is irrelevant; its existence is the signal.

The protocol:

1. Sub-agent performs its task
2. Sub-agent writes the structured JSON output file
3. Sub-agent writes the `.done` marker as its final file operation
4. Orchestrator (or `wait_for_done.py`) detects the marker
5. Orchestrator reads the JSON output file

Without this protocol, the orchestrator could read a partially-written JSON file, producing parse errors or incomplete data.

§5.5.4 Stale Marker Management

Before dispatching any sub-agent, the orchestrator MUST clear stale `.done` markers from the relevant output path. This prevents the orchestrator from reading output from a previous dispatch cycle.

```
# Before implementation dispatch
mkdir -p <impl-dir>/.impl-work/<spec-name>/ && rm -f <impl-dir>/.impl-work/<spec-name>/summary.done

# Before verification dispatch (clear all fragment markers)
mkdir -p <impl-dir>/.impl-verification/<spec-name>/fragments/ && rm -f
    ↪ <impl-dir>/.impl-verification/<spec-name>/fragments/*.done
```

Failure to clear stale markers can cause the orchestrator to read output from a previous agent, producing incorrect state updates. This is a silent failure — no error is raised; the data is simply wrong.

---

§5.6 Parallel Dispatch and Coordination

The skill uses parallel dispatch in two contexts: verification (many agents, one per requirement) and implementation (limited parallelism to manage context load). Coordination relies on deterministic polling, not on conversational interaction with sub-agents.

§5.6.1 Verification Dispatch (Fully Parallel)

During Phase 3, the orchestrator dispatches one verification sub-agent per requirement, all running in background. A medium-sized specification with 30-60 requirements produces 30-60 parallel agents. This is intended and correct — the per-requirement granularity is a hard rule (see §5.7.1).

All verification agents MUST be dispatched with `run_in_background: true`. The orchestrator MUST NOT call `TaskOutput` on any verification agent. Calling `TaskOutput` blocks the orchestrator and loads the agent's conversational response into the main context — violating both the parallelism model and the output-to-disk pattern.

§5.6.2 Implementation Dispatch (Limited Parallelism)

During Phase 2, the orchestrator MUST limit concurrent heavy-context sub-agents to a maximum of 2 at a time. Implementation agents consume more resources than verification agents (they modify files, run tools, produce larger outputs), and exceeding this limit can degrade overall throughput and produce file-write conflicts. This is a hard limit, not a guideline.

The orchestrator MAY dispatch lightweight agents (haiku-class, boilerplate tasks) alongside a heavy-context agent without counting them against the limit.

§5.6.3 Coordination via wait_for_done.py

The `wait_for_done.py` script provides deterministic polling for `.done` markers. It replaces the need to call `TaskOutput` on individual agents. The `$IMPL_PYTHON` and `$IMPL_TOOLS_DIR` variables are resolved during Common Initialization (§3.0.1) and MUST NOT be redefined locally.

```
"$IMPL_PYTHON" "$IMPL_TOOLS_DIR/wait_for_done.py" \
  --dir <impl-dir>/.impl-verification/<spec-name>/fragments/ \
  --count <number of requirements dispatched>
```

The script polls the specified directory until the expected number of `.done` files appear. It returns when all agents have completed or after a timeout. This is a deterministic coordination mechanism — no LLM inference is involved in the waiting, counting, or completion detection.

Timeout contract: The default timeout is 600 seconds. Exit code 0 indicates all expected `.done` markers were found (success). Exit code 1 indicates timeout — stderr will list the missing markers. There is no dead-agent detection beyond the timeout; the orchestrator cannot distinguish between a slow agent and a crashed agent. This is a known limitation — the timeout serves as the only liveness boundary.

§5.6.4 Coordination Rules

The orchestrator MUST follow these coordination rules:

1. Clear stale `.done` markers before each dispatch batch
2. Dispatch all agents in the batch with `run_in_background: true`
3. Use `wait_for_done.py` to wait for completion — never `TaskOutput`
4. After `wait_for_done.py` returns, read output JSON files from disk
5. For verification: pass fragments to `verify_report.py` for deterministic assembly
6. Never assume an agent has completed based on elapsed time — always check `.done` markers

---

§5.7 Prompt Templates

The skill provides prompt templates in the `prompts/` directory. Each template specifies the task, the expected input (spec sections, code paths), the output format (JSON schema), and the `.done` marker path. The orchestrator MUST use these templates when dispatching sub-agents — ad-hoc prompts are not permitted for standard operations.

§5.7.1 Template Catalogue

Template	File	Purpose	Model
Single-file implementation	`prompts/implement-single.md`	Implement requirements from a single-file spec	Per task complexity
Multi-file implementation	`prompts/implement-mul`	Implement requirements from a multi-file spec with section files	Per size routing (§5.3.2)
Write tests	`prompts/write-tests.md`	Write tests after implementation (standard workflow)	`sonnet` default
TDD write tests	`prompts/tdd-write-tes`	Write tests before implementation (TDD workflow)	`sonnet` default
TDD implement	`prompts/tdd-implement.md`	Implement code to make failing TDD tests pass	`sonnet` default
Verify requirement	`prompts/verify-requir`	Verify one requirement against the codebase	`opus` always
Re-verify requirement	`prompts/reverify-requirement.md`	Re-check a previously flagged V-item	`opus` always
Fix issue	`prompts/fix-issue.md`	Fix a problem found during implementation	`opus` always
Fix verification gap	`prompts/fix-verificatio-gap.md`	Fix a gap identified during verification	`opus` always

Template	File	Purpose	Model
Spec compliance check	`prompts/spec-compliar`	Post-task lightweight compliance audit	`sonnet` default
Opus escalation review	`prompts/opus-escalation-review.md`	Review sonnet agent's work when DIGEST signals complexity	`opus` always

Required template — opus escalation review: The opus escalation review template (`prompts/opus-escalation-review.md`) MUST exist. It is dispatched when a sonnet agent's DIGEST triggers complexity escalation (§5.4.3 step 4). The template MUST include the following inputs:

- The DIGEST category (from §5.4.2) that triggered escalation
- The sonnet agent's output: code changes and summary from `summary.json`
- The spec section(s) being reviewed (file paths or quoted text)
- Instructions to focus the review on the flagged complexity area
- Output format: `summary.json` with the same schema as other implementation agents, including a `digest` field, written to the same disk path (overwriting the sonnet's output)
- Completion marker: a `.done` marker, following the same protocol as all other sub-agents (§5.5.3)

  Note: This template does not yet exist in the implementation. It is a known action item that MUST be created before the opus escalation pathway is functional.

§5.7.2 Template Structure

Every prompt template follows a consistent structure:

1. **Task description**: What the agent is being asked to do
2. **Input sections**: Spec text (quoted or via file path), implementation hints, context
3. **Instructions**: Specific guidance on how to perform the task
4. **Self-review section** (implementation templates): Pre-completion quality check
5. **Output specification**: Exact JSON schema and disk path for structured output
6. **Completion marker**: The `.done` marker path and instruction that it MUST be the last file written
7. **Conversational response**: Instruction to respond with a minimal acknowledgment (e.g., "Done.")

§5.7.3 Template Usage Rules

- The orchestrator MUST fill in all template placeholders (spec sections, file paths, section references) before dispatch
- The orchestrator MUST NOT modify the JSON output schema — sub-agents produce output in the documented format, and downstream tools depend on that format
- For multi-file specs, the orchestrator SHOULD pass the section file path rather than quoting the full spec text in the prompt — this avoids duplicating large spec sections in the orchestrator's context
- Each template specifies its `.done` marker path — the orchestrator MUST clear stale markers at that path before dispatch (see §5.5.4)

---

§5.8 Delegation Boundaries

The division between what the orchestrator keeps and what it delegates is not arbitrary. It follows directly from the skill's design principles: the orchestrator manages state and makes decisions; sub-agents perform focused work with fresh context.

§5.8.1 What MUST Be Delegated

Activity	Why It Must Be Delegated
Implementation (writing code)	Fresh context ensures spec fidelity; prevents orchestrator context bloat
Test writing	Especially in TDD mode — isolation from implementation is the mechanism, not a side-effect
Requirement verification	Independent judgment requires independent context
Gap fixing	Fresh context avoids inheriting the original implementer's misunderstanding
Spec compliance checking	Independent review from a context that has not been influenced by the implementation process

§5.8.2 What MUST Stay in the Orchestrator

Activity	Why It Must Stay in Main
Planning (Phase 1)	Requires developer interaction and global project understanding
Tracker reading and updating	Tracker is the orchestrator's state — sub-agents MUST NOT modify it
Task sequencing and prioritisation	Requires awareness of the full requirements matrix and dependencies
Model selection decisions	Requires the structural index and task complexity assessment
DIGEST escalation decisions	The orchestrator checks the digest; the sub-agent only produces it
Developer communication	Status updates, ambiguity resolution, decision requests
Test execution and result interpretation	The orchestrator runs tests and decides how to proceed based on results. Test runner output is captured to a file (e.g., `test-output.txt`) using shell redirection; the orchestrator reads only the exit code and the final summary lines, not the full output, to avoid loading verbose test output into the main context.
Verification report interpretation	The orchestrator reads the assembled report and plans remediation

§5.8.3 The Boundary Principle

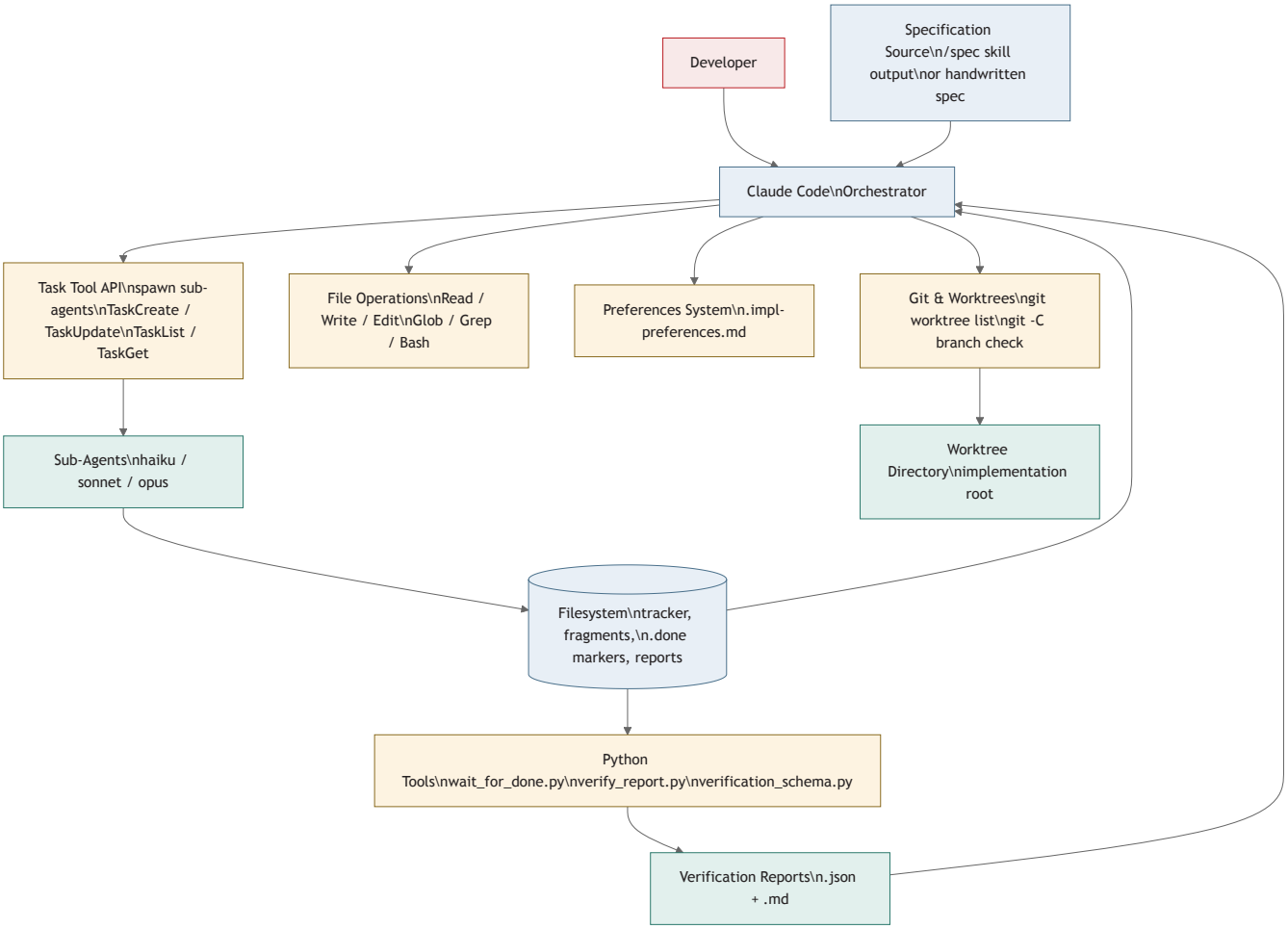
The boundary can be stated simply: **the orchestrator reasons about what to do; sub-agents do it.** If a task requires judgment about which work to perform next, it belongs in the orchestrator. If a task requires reading specification text and producing code or findings, it belongs in a sub-agent.

This boundary is enforced by the output-to-disk pattern (§5.5). Because sub-agents write to disk and the orchestrator reads from disk, the communication channel between them is structured, auditable, and immune to the conversational drift that occurs when the orchestrator tries to both coordinate and implement in the same context window.

§6 Integrations & Dependencies

§6.0 Integration Architecture Overview

The `/implement` skill is not a standalone system. It coordinates a small set of tightly coupled integrations, each playing a distinct role in the implementation pipeline. The architecture separates LLM inference from deterministic assembly: Claude tools handle reasoning and delegation; Python tools handle aggregation and polling; the filesystem is the shared state bus.



The key data flows are:

- Sub-agents write structured JSON fragments to disk. They do not return results conversationally.
- The orchestrator never reads sub-agent conversational output. It reads files.
- Python tools perform all aggregation deterministically — no LLM inference in the assembly pipeline.
- The filesystem is the only channel between the orchestrator, sub-agents, and Python tools.

---

§6.1 Claude Code Platform

What It Provides

The Claude Code platform is the execution environment for the skill. All skill behaviour runs within Claude Code's tool API surface.

Tool	Purpose in Skill
`Task`	Spawn sub-agents with specific models, instructions, and `run_in_background` flag
`TaskCreate`	Create persistent task records that survive context compaction
`TaskUpdate`	Update task status as implementation progresses
`TaskList`	List all active tasks — used during recovery and status checks
`TaskGet`	Read a specific task's details
`Read`	Read spec sections, tracker files, summary JSON, verification fragments
`Write`	Write tracker files, preference files, completion markers
`Edit`	Update tracker fields in place
`Glob`	Discover tracker files, verification fragments, spec sections
`Grep`	Search implementation files for references and patterns
`Bash`	Run tests, linters, git commands, Python tools, `wc -c` for size routing

Required or Optional

MUST. The entire skill is a Claude Code skill and cannot function outside this platform.

What Happens If Unavailable

The skill does not exist. There is no fallback environment.

Version Pinning

There is no version pinning. The skill uses whatever tool API Claude Code provides at runtime. If a tool interface changes, the skill's behaviour may be affected — but no version constraint is declared.

Sub-Agent Spawning

Sub-agents are spawned via the `Task` tool. Two flags are used:

- `model`: selects the Claude tier (see §6.5)
- `run_in_background`: set to `true` for verification agents to enable parallel dispatch

The orchestrator MUST NOT call `TaskOutput` on any sub-agent (verification, implementation, or otherwise). It waits for `.done` marker files on disk and reads structured JSON output files instead. This is intentional and applies to all agent types: calling `TaskOutput` consumes orchestrator context window, whereas reading a marker file and a compact JSON file costs far less. See §5.5 for the full output-to-disk pattern.

---

§6.2 Git & Worktrees

What It Provides

Git worktree support allows multiple concurrent implementations to proceed in isolation. Each implementation operates in its own working directory on its own branch, with its own tracker file. This enables a developer to implement two features from the same codebase simultaneously without branch conflicts.

Integration Points

Operation	Command	Phase
Detect worktree intent	Read spec/brief for path/branch mentions	Phase 1
Validate worktree exists	`git worktree list` from project root (or from within a worktree)	Phase 1
Confirm active branch	`git -C <worktree-path> branch --show-current`	Phase 1
Re-validate on resume	Same as above	Phase 5
Worktree-aware verification	Resolve all paths relative to worktree	Phase 3

The worktree path is stored in the tracker's `**Worktree**` field. All subsequent operations — tracker creation, file edits, test runs, verification artifacts — resolve relative to the worktree path, not the main conversation's working directory.

Worktree Discovery in Phase 1

Worktree detection is intentionally flexible. The orchestrator reads the specification or brief document and extracts the implementation directory intent from natural language — it does not require a specific section name or field format. A spec might say "implement in `/path/to/worktree`" or "work on the `feature/billing` branch" and both would be detected.

Tracker files are named `.impl-tracker-<spec-basename>.md` and created in the implementation directory (the worktree, if one was detected). The `**Worktree**` field enables tracker discovery across multiple directories: when listing trackers, the skill checks both the current directory and any worktree paths recorded in existing trackers.

Worktree path discovery: To discover all known worktree paths, the orchestrator runs:

```
git worktree list --porcelain
```

from the project root. This produces a machine-readable list of all registered worktrees (main and linked). The orchestrator then scans each listed worktree directory for `.impl-tracker-*.md` files. This covers cases where an in-progress implementation exists in a worktree that is not the current working directory.

Required or Optional

SHOULD. Git worktree support is optional — if no worktree is specified, the skill works in the current directory. However, for projects with concurrent feature work, worktrees are strongly recommended and the skill is designed to exploit them.

What Happens If Git Is Unavailable

If `git worktree list` fails, the skill warns the user and falls back to using the current directory. Worktree validation is a pre-condition for worktree-aware operation, not for operation in general.

What Happens If Worktree Validation Fails on Resume

If Phase 5 (`/implement continue`) detects that the worktree path recorded in the tracker no longer exists or is on the wrong branch, the orchestrator warns the user and asks whether to:

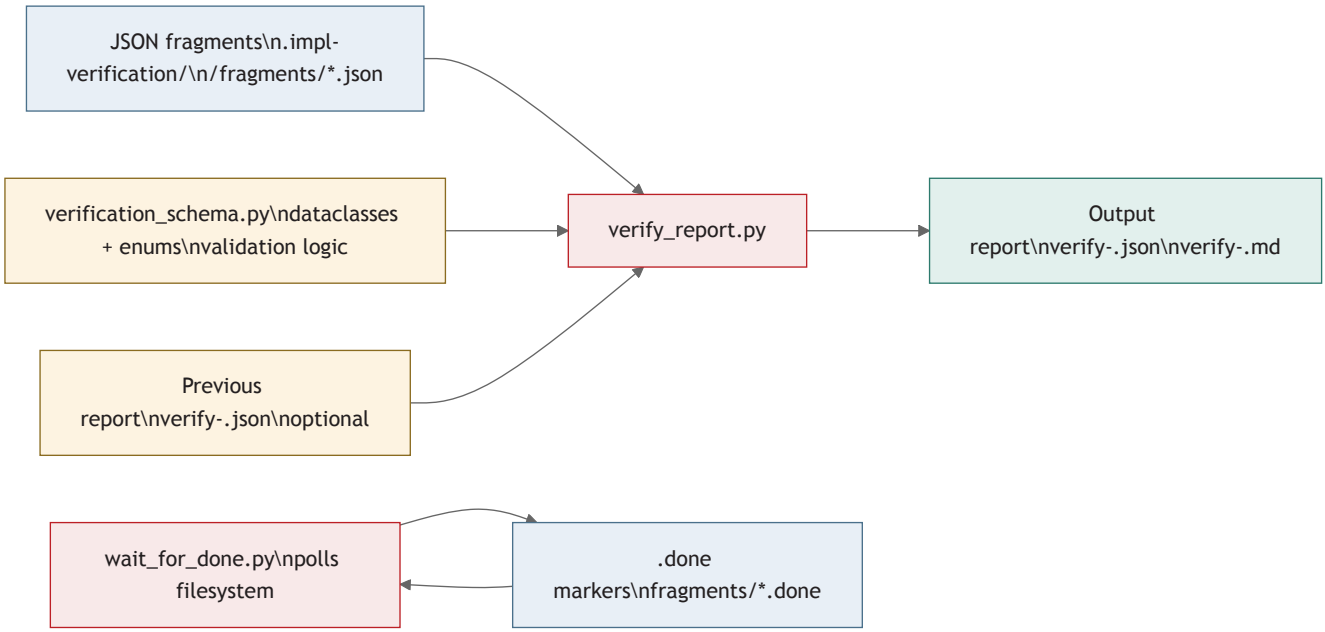
- Abort and resolve the worktree state
- Proceed using the current directory instead

---

§6.3 Python Tooling

What It Provides

Three Python scripts at `tools/` in the skill repository perform deterministic tasks that are better handled by code than by LLM inference.



verify_report.py

Purpose: Assembles individual verification JSON fragments produced by sub-agents into a single structured report with statistics, MoSCoW breakdown, priority gaps, and markdown rendering.

Invocation:

```
"$IMPL_PYTHON" "$IMPL_TOOLS_DIR/verify_report.py" \
  --fragments-dir <impl-dir>/.impl-verification/<spec-name>/fragments/ \
  --spec-path <spec-path> \
  --impl-path <impl-dir> \
  --project-name "<spec-name>" \
  --output <impl-dir>/.impl-verification/<spec-name>/verify-<date>.json \
  [--previous <impl-dir>/.impl-verification/<spec-name>/verify-<prev-date>.json] \
  [--spec-version "1.0"] \
```

```
    [--verbose]
```

Outputs (always both):

- `verify-<date>.json` — machine-readable report, used as `--previous` in re-verification
- `verify-<date>.md` — human-readable report rendered by `render_markdown()`

Re-verification mode: When `--previous` is supplied, the tool maps V-item IDs from the previous report to new findings by `section_ref`, carries forward IDs for matched items, assigns fresh sequential IDs to new items, and computes a `ResolutionSummary` with fixed/partially_fixed/not_fixed/regressed counts.

Error handling: `verify_report.py` exits with code `0` on success and non-zero on failure. Common failure modes include: missing or empty fragments directory, invalid JSON in a fragment file, and missing output directory. On failure, the tool writes a partial or no report. The orchestrator MUST check the exit code after invocation and surface any non-zero exit to the user with the stderr output, rather than silently proceeding with a missing or incomplete report.

Determinism guarantee: Given the same set of fragments and the same previous report, the output is bit-identical. No LLM inference occurs in this tool.

Output directory responsibility: The orchestrator is responsible for creating both the fragments directory (`.impl-verification/<spec-name>/fragments/`) and the report output directory before invoking verification sub-agents or `verify_report.py` (e.g., via `mkdir -p`). Neither the sub-agents nor the Python tools create directories — they will fail if the target paths do not exist.

wait_for_done.py

Purpose: Blocks until a specified number of `.done` marker files appear in a directory, or until specific named files appear. Replaces manual sleep/poll loops that would consume orchestrator context window.

Invocation (two modes):

```
    # Wait for N .done files in a directory
    "$IMPL_PYTHON" "$IMPL_TOOLS_DIR/wait_for_done.py" --dir <fragments-dir> --count <N> [--timeout 600] [--interval 2]

    # Wait for specific named files
    "$IMPL_PYTHON" "$IMPL_TOOLS_DIR/wait_for_done.py" --files path/a.done path/b.done [--timeout 600]
```

Exit codes: `0` if all markers found; `1` if timeout reached.

Default timeout: 600 seconds (10 minutes). For large verification runs (40+ agents), this may need adjustment.

Timeout sizing guidance: For a spec with N verification agents, a reasonable timeout is:

```
    timeout = max(600, N * 30)   # 30 seconds per agent, minimum 600s
```

Pass a custom value with the `--timeout` flag:

```
    "$IMPL_PYTHON" "$IMPL_TOOLS_DIR/wait_for_done.py" --dir <fragments-dir> --count <N> --timeout <seconds>
```

This is guidance, not an enforced constraint — the orchestrator may use any reasonable timeout based on observed agent latency or known task complexity.

Why a Python script and not a Bash loop: A bash poll loop would produce repeated tool call outputs that accumulate in the orchestrator's context window. The Python script blocks once, produces one line of output, and exits. This is a context efficiency design choice.

verification_schema.py

Purpose: Defines the dataclasses, enums, validation logic, and rendering functions used by `verify_report.py`. Not invoked directly.

Key types:

Type	Role
`Finding`	One verified requirement — the unit of verification
`Status`	`implemented`, `partial`, `not_implemented`, `na`
`MoSCoW`	`MUST`, `SHOULD`, `COULD`, `WONT`
`TestCoverage`	`full`, `partial`, `none`
`Resolution`	`fixed`, `partially_fixed`, `not_fixed`, `regressed` (re-verification only)
`VerificationReport`	The assembled report with metadata, findings, statistics, priority gaps
`PriorityGap`	A gap finding with computed priority: `high`, `medium`, `low`
`ResolutionSummary`	Delta between two verification runs

Fragment validation: Each JSON fragment produced by a verification sub-agent is validated against required fields and enum values before loading. Hard errors raise `SchemaError` and abort assembly. Consistency warnings (e.g., status is `implemented` but `missing_implementation` is non-empty) are logged but do not abort.

V-item ID assignment: Initial verification assigns `V1`, `V2`, ... in `fragment_id` sort order (deterministic). Re-verification carries forward existing V-item IDs by matching `section_ref`. New items not in the previous report receive the next available sequential ID. V-item IDs are permanent — once assigned, they are never reused or changed across runs.

Tool Discovery at Runtime

The orchestrator resolves the tools directory during Common Initialization (see §3.0), before entering any phase. Three variables are defined once and reused throughout all phases:

```
IMPL_REPO_DIR="$(cd "$(dirname "$(realpath ~/.claude/skills/implement/SKILL.md)")/../.." && pwd)"
IMPL_TOOLS_DIR="$IMPL_REPO_DIR/tools"
IMPL_PYTHON="$IMPL_REPO_DIR/.venv/bin/python"
```

Key design decisions:

- **Prefixed variable names** (`IMPL_` prefix): Prevents namespace collisions when multiple skills are loaded in the same session (e.g., `/spec` uses `SPEC_REPO_DIR`).
- **Venv Python** (`$IMPL_PYTHON`): Uses the skill repo's virtual environment Python rather than the system `python3`. This ensures a consistent Python version across environments. The tools currently use only the standard library; the venv provides a path for future pip dependencies if needed without changing invocation patterns.
- **Resolved once, used everywhere**: Previous versions resolved tool paths ad-hoc in individual phases (e.g., verification). Centralising resolution in Common Initialization eliminates inconsistent variable names and ensures all phases use the same paths.

Platform note: `realpath` is available on both macOS (BSD) and Linux without additional dependencies. The `cd ... && pwd` wrapper normalises the path, eliminating literal `/../..` segments from the resulting variable. The

`dirname` traverses two parent directories because SKILL.md is installed at `<repo>/skills/implement/SKILL.md` — two levels below the repo root.

Venv existence check: After resolving `$IMPL_PYTHON`, the skill MUST verify the path exists (`test -x "$IMPL_PYTHON"`). If it does not exist, the skill MUST warn the user: "The implement skill's Python venv is missing. Run `python3 -m venv .venv` in the skill repository root." See FR-0.3.

The tools directory is resolved relative to the installed SKILL.md location, making it robust to different installation paths.

Required or Optional

MUST for Phase 3 (Verification). The Python tools are hard dependencies for verification. There is no fallback — manual assembly of verification fragments by the LLM is explicitly prohibited because it would be non-deterministic and context-expensive.

SHOULD have Python 3 available in the execution environment. No version constraint is declared beyond Python 3. The tools currently use only the standard library (the venv ensures version consistency and provides a path for future dependencies).

What Happens If Python Is Unavailable

Verification cannot proceed. The orchestrator should detect the failure from `bash` exit code and inform the user that Python 3 is required for verification.

---

§6.4 Specification Sources

What It Provides

The `/implement` skill consumes specification documents. It does not produce them. The primary intended source is the `/spec` skill, but any well-structured markdown document works.

Integration with /spec Output

The `/spec` skill produces a master `spec.md` with a table of contents and a `sections/` directory containing one file per section. This structure triggers the multi-file parsing path in Phase 1:

1. The orchestrator reads only the master spec's table of contents — not all section files.
2. It runs `wc -c` on all section files to build a structural index with estimated token counts.
3. Sub-agents receive individual section files in their task prompts and read them independently.

This design prevents the orchestrator from loading a large specification into its own context window. Sub-agents read sections; the orchestrator indexes them.

The `/spec` skill also uses `<!-- EXPANDED: -->` markers in the master spec to indicate that a section has a breakout file. Phase 1 detects these markers as an alternative signal for multi-file specs.

EXPANDED marker syntax: The full form is:

```
<!-- EXPANDED: sections/02-architecture.md -->
```

The path is relative to the spec directory. The marker replaces the section content inline in the master spec — the actual content lives in the referenced file. Readers (and Phase 1) should follow the path to retrieve the full section text.

Section References

Specifications produced by `/spec` use `§N.M` section references (e.g., `§2.4`, `§9.1`). These references are:

- Used as keys in the tracker's Requirements Matrix
- Included in task subjects and descriptions
- Used by verification sub-agents to identify which requirement they are checking
- Used by `verification_schema.py` to match V-items across re-verification runs (via `section_ref` field)

Section references are described as **stable anchors** — they survive context compaction because they are short strings written to persistent files, not kept only in the LLM's context.

Graceful Degradation Without /spec

The skill does not require `/spec`-style output. Any markdown specification document works, including:

- Hand-written documents with numbered headings
- Business process documents
- Requirements documents with `##` section structure

Without `§N.M` references, the skill extracts whatever section identifiers the document provides (numbered headings, section numbers, topic names). Verification still functions but V-item-to-section matching in re-verification is less precise.

Section Identifier Extraction for Non-/spec Documents

When a document lacks `/spec`-style `§N.M` markers, the orchestrator uses the following heuristic extraction approach:

1. **Scan for heading patterns** in order of preference:
   - `## Section N` or `### Section N.M` (explicit "Section" keyword)
   - `§N.M` or `§N` anywhere in a heading
   - Numbered headings: `## 1.`, `## 2.3`, `### 4.2.1`
   - Bare headings: `## Introduction`, `## Architecture`
2. **Extract with hierarchy**: Identifiers carry their heading level context. A `###` under a `##` is treated as a child section.
3. **Fallback for unstructured documents**: If no numeric or `§`-prefixed identifiers are found, headings are assigned sequential labels: `H1`, `H2`, `H3`, ... in document order.

This extraction is heuristic. Hand-written specs with no consistent structure produce coarser-grained verification items — a single `H1`-level identifier may cover a large portion of the document, reducing the precision of per-requirement tracking.

STRUCT Awareness

If a `.spec-tracker-*.md` file exists in the spec's directory (indicating the spec was produced by `/spec`), Phase 1 checks for a `## Pending Structural Changes` section. If pending structural issues are present, the user is warned before planning proceeds. This is a SHOULD-level integration — the skill can proceed without it.

Required or Optional

MUST have some specification document as input. The `/spec` skill output is SHOULD — it provides the best experience (section references, structural index, multi-file parsing) but is not a hard dependency.

What Happens If the Spec Is Missing or Unreadable

Phase 1 cannot proceed. The orchestrator will report the read failure and ask the user to provide a valid path.

---

§6.5 Model Tier Requirements

What It Provides

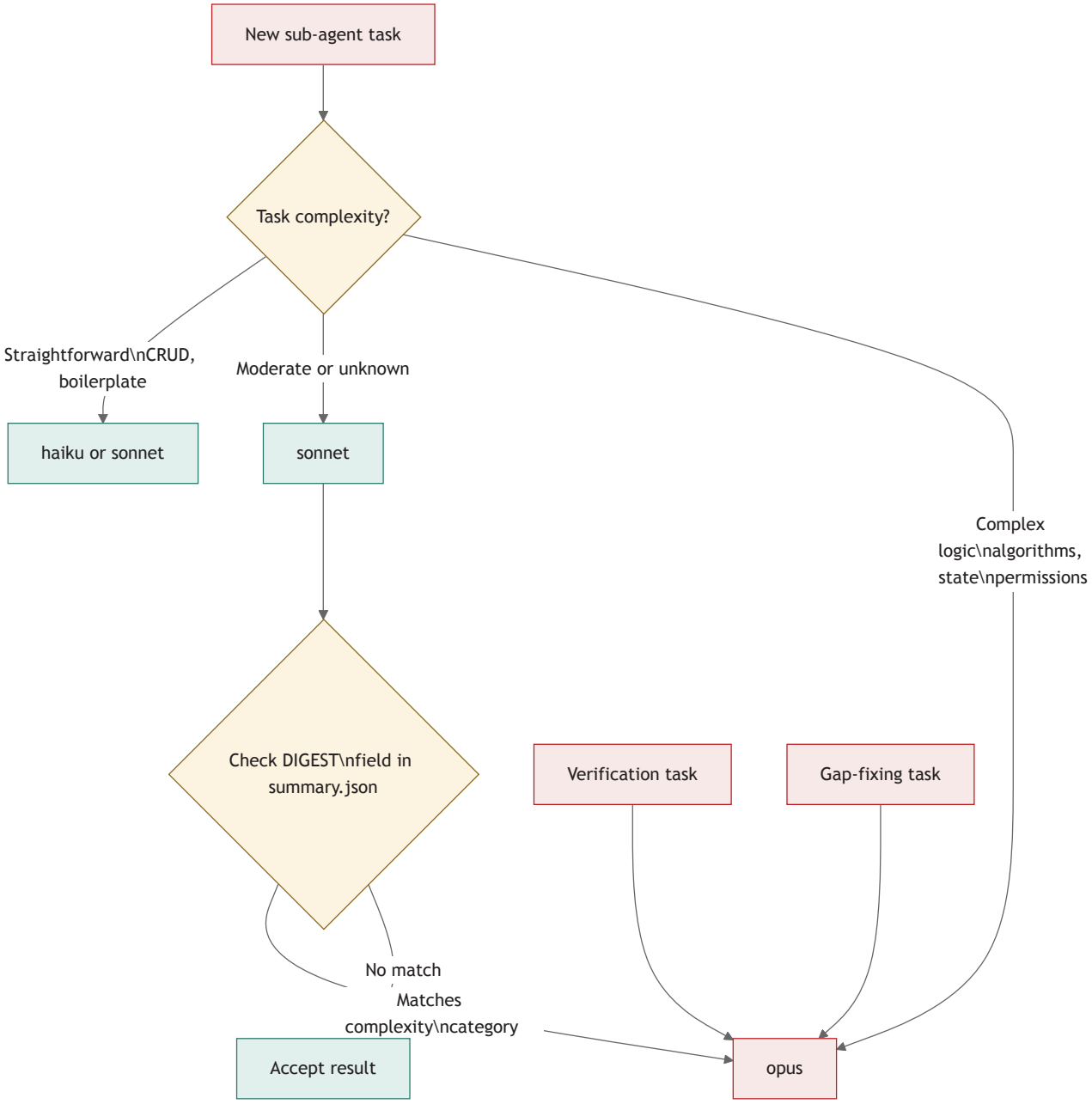
The skill uses three Claude model tiers with different capability/cost profiles. Model selection is per-task, not per-session.

Tier	Tier Name	Used For
Fast/cheap	`haiku`	Simple, boilerplate, or purely additive tasks (e.g., adding imports, creating stub files, boilerplate config)
Mid-tier	`sonnet`	Moderate implementation tasks; default for multi-file spec routing; any task with non-trivial logic
Capable	`opus`	Complex logic, algorithms, state management, all verification, all gap fixing

Haiku vs sonnet decision: When in doubt, prefer `sonnet`. The `haiku` tier is reserved for tasks where the orchestrator has high confidence that no reasoning or judgment is required — pure boilerplate generation, file copying, or trivial additions. If a task involves any conditional logic, interpretation of requirements, or multi-step reasoning, use `sonnet` as the minimum tier.

Note on tier names: `haiku`, `sonnet`, and `opus` are abstract tier names used by the skill — they are not literal API model ID strings. The Claude Code `model` parameter in `Task()` calls maps these names to the current model versions at runtime. The skill does not pin specific model version strings and does not need to be updated when new model versions are released.

Selection Rules



DIGEST escalation (for sonnet agents implementing multi-file specs): After a sonnet sub-agent completes, the orchestrator reads `summary.json` from `.impl-work/<spec-name>/summary.json` and extracts the `digest.complexity` sub-field (see §5.4.1 for the canonical `digest` schema, which is a structured JSON object with `entities`, `patterns`, and `complexity` sub-fields). If `digest.complexity` matches any of the complexity categories below, an opus review agent is dispatched. This escalation is MANDATORY — it is not discretionary.

DIGEST error handling: If the `digest` field is absent, malformed, or missing the `complexity` sub-field in `summary.json`, the orchestrator SHOULD log a warning and skip escalation for that agent (fail-open). A missing or unreadable digest does not block the workflow — the implementation result is accepted as-is without an opus review pass.

> **Convenience copy** of the canonical complexity category table in §5.4.2. See §5.4.2 for the authoritative source. In case of conflict, §5.4.2 takes precedence.

Category	DIGEST signals
Algorithms	"algorithm", "calculation", "formula", "heuristic"
State machines	"state machine", "state transition", "lifecycle"
Permission/auth	"permission", "role inheritance", "RBAC", "access control"
Complex business rules	"conditional", "override", "exception", "cascading"
Cross-cutting	"affects all", "global constraint", "system-wide"

Size-based routing (multi-file specs only):

Section estimated tokens	Model	Grouping
< 5k tokens	`sonnet`	Group small sections (see below)
5k–20k tokens	`sonnet`	1 section per agent
> 20k tokens	`opus`	1 section per agent

Estimated tokens are computed as `file_size_bytes / 4` using output from `wc -c`.

Small section grouping algorithm: Contiguous sections estimated at < 5k tokens each are grouped greedily — sections are accumulated in document order until adding the next section would push the group's total estimated token count over 5k, at which point a new group starts. A maximum of 3 sections per group is enforced regardless of token count. Each group is dispatched to a single sonnet agent.

Required or Optional

MUST. All three model tiers (haiku, sonnet, opus) MUST be available. There is no fallback if a tier is unavailable — the skill will fail when it attempts to spawn an agent with that model.

This is a hard dependency by design: the cost-conscious tiering strategy only works if cheaper tiers are available for appropriate tasks, and verification correctness depends on opus being available for verification and gap-fixing.

What Happens If a Tier Is Unavailable

If a requested model tier is unavailable, the `Task` tool call will fail. The orchestrator should surface the failure to the user. There is no automatic fallback to another tier — the user must resolve the availability issue.

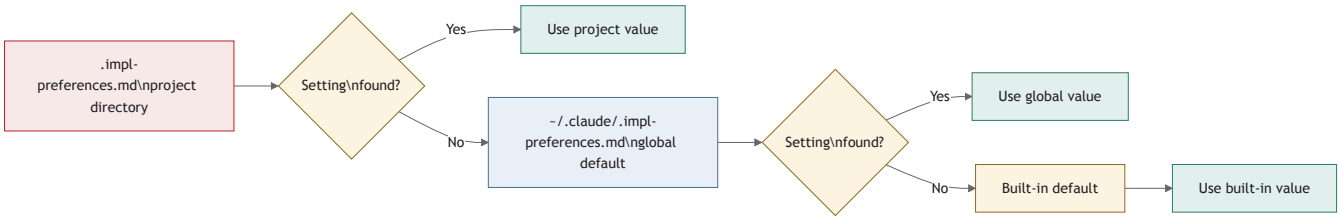
---

§6.6 Preferences System

What It Provides

The preferences system allows users to configure skill behaviour across sessions and across projects without modifying skill files. It is a simple markdown-file-based configuration layer.

File Locations and Lookup Order



Lookup order: project file → global file → built-in default. Project settings override global settings.

File Format

```
# Implementation Preferences

## Workflow
- **tdd-mode**: on
```

Available Preferences

Preference	Values	Built-in Default	Description
`tdd-mode`	`on`, `off`, `ask`	`on`	Controls TDD workflow activation

`tdd-mode` behaviour:

- `on` — Always use TDD (tests written by a dedicated sub-agent from the spec before implementation code is written)
- `off` — Always use standard mode (implementation first, tests after)
- `ask` — Prompt the user during Phase 1 planning to choose per-implementation

/implement config Command

The `/implement config` command provides a read/write interface to the preferences system from within a Claude Code session. It does not require the user to manually edit markdown files.

Command	Effect
`/implement config`	Read and display the current effective preferences from all three sources
`/implement config tdd on|off|ask`	Write the setting to `.impl-preferences.md` in the project directory
`/implement config --global tdd on|off|ask`	Write the setting to `~/.claude/.impl-preferences.md`

When writing a preference:

1. Read the existing preferences file (create it if it does not exist)
2. Update the specified value in place
3. Write the file back
4. Confirm the change to the user

Required or Optional

COULD. The preferences system is entirely optional. If no preference files exist, the skill operates using built-in defaults (`tdd-mode: on`). Deleting preference files restores default behaviour.

The preferences system MUST NOT block operation if files are missing, malformed, or unreadable. Any read failure falls through to the next level in the lookup chain. A malformed preferences file is treated as if it does not exist — the skill logs a warning and continues with the next level.

What Happens If Preference Files Are Missing

The skill uses built-in defaults. This is the normal state for first-time users.

What Happens If the Global Preferences File Is Inaccessible

The skill falls back to built-in defaults for any settings not found in the project-level file. Home directory access is assumed but not enforced as a hard dependency.

§7 Non-Functional Requirements

Non-functional requirements govern how the `/implement` skill performs across dimensions that are orthogonal to individual features: how efficiently it uses context, how much it costs to run, how well it scales, how reliably it recovers from failure, and how predictably its assembly pipeline behaves. These requirements are not aspirational — they are structural constraints that shaped the skill's architecture from the ground up.

---

§7.1 Context Window Efficiency

NFR-CTX-01: Skill entry point MUST fit within a minimal context budget

Requirement: The skill's primary entry point (`SKILL.md`) MUST be structured so that loading it does not consume a disproportionate share of the orchestrator's context window. Phase reference files MUST be loaded on demand, not pre-loaded in full at skill activation. `SKILL.md` SHOULD remain under 400 lines (approximately 12,000–16,000 tokens at typical prose density). Growth beyond this threshold indicates the file needs restructuring.

Rationale: The orchestrator's context window is the primary resource constraint in long implementation sessions. A skill that loads thousands of lines of reference material at startup leaves less room for specification content, implementation status, and developer interaction — the things that actually matter. Bloated skill files also increase the risk of the orchestrator itself suffering from the U-shaped attention degradation documented in §1.1.

How the skill meets it: `SKILL.md` was restructured from approximately 1,400 lines to approximately 296 lines as a deliberate context optimisation. Phase reference files (planning, implementation, verification, etc.) are referenced by path and loaded only when the relevant phase begins. The skill's own structure embodies the principle it enforces for specifications.

NFR-CTX-02: Sub-agents MUST write structured output to disk, not return conversational responses

Requirement: Sub-agents MUST write their results as structured artefacts to disk (e.g., JSON summary files, `.done` markers, verification fragments). The orchestrator reads these artefacts from disk rather than parsing the sub-agent's conversational reply.

Rationale: Conversational sub-agent responses flow back into the orchestrator's context window, consuming tokens for content the orchestrator may only need to sample lightly (e.g., "did it succeed? any concerns?"). Structured artefacts on disk allow the orchestrator to read targeted fields — `status`, `concerns`, `digest` — without ingesting the full agent reply.

How the skill meets it: All sub-agent prompt templates instruct the agent to write a structured `summary.json` to `.impl-work/<spec-name>/summary.json` and to place a `.done` marker when complete. The orchestrator reads these files directly; it does not rely on the agent's conversational output for decision-making.

NFR-CTX-03: Parallel heavy-context phases MUST be limited to two concurrent draft agents

Requirement: Implementation sub-agents MUST NOT exceed 2 concurrent heavy agents (sonnet or opus tier). Verification sub-agents are exempt from this cap — they are lightweight, uniform, and dispatched in parallel by design.

Rationale: Each sub-agent spawned consumes model capacity with a full context load. Spawning many heavy-context agents simultaneously provides diminishing throughput returns while increasing the risk of resource

contention and degraded output quality. Verification agents operate on a per-requirement basis with much smaller context loads and are structured for safe parallelism.

How the skill meets it: Orchestrator guidance in the planning phase caps concurrent draft agents at two for large-section work. Verification agents may be parallelised freely up to platform limits (see §7.3).

NFR-CTX-04: The tracker file MUST maintain a minimal token footprint while preserving recoverability

Requirement: The implementation tracker file MUST encode all state needed for session recovery in a compact, structured format. It MUST NOT accumulate unbounded content (e.g., full implementation notes, code excerpts, or verbose logs) that would make it expensive to load at the start of every orchestrator action.

Rationale: The tracker is read at the beginning of every orchestrator decision cycle. If it grows without bound, it imposes an ever-increasing context tax on every step of the workflow. At the same time, it must contain enough information for full recovery after context compaction — a tension that demands careful structural discipline.

How the skill meets it: The tracker uses a fixed-schema table format for requirement status, section references, and phase state. Free-text fields are bounded: Implementation Log entries SHOULD NOT exceed 200 words; the Specification Summary SHOULD NOT exceed 500 words. These are advisory targets rather than enforced limits, but consistent compliance is expected to prevent tracker growth from imposing context overhead. The schema is designed to convey maximum recovery information in minimum tokens: a structured table row encodes requirement ID, status, section reference, and any blocking issue in a single line.

---

§7.2 Cost Management

NFR-COST-01: The skill MUST apply model tiering based on task complexity

Requirement: The skill MUST select model tier based on task characteristics. Haiku or Sonnet MUST be used for boilerplate and routine tasks. Sonnet MUST be used for standard implementation. Opus MUST be used for verification, gap fixing, and complex reasoning tasks.

Rationale: Opus is the most capable and most expensive tier. Applying it uniformly across all sub-agent tasks would make the skill economically impractical for real-world projects with hundreds of requirements. Conversely, using cheaper models for verification risks missing subtle specification violations — which defeats the skill's entire purpose. Model selection must be a function of where correctness risk is highest.

How the skill meets it: The sub-agent strategy reference defines explicit routing rules:

Task	Model
Boilerplate, simple CRUD, adding fields	Haiku or Sonnet
Standard implementation	Sonnet
Verification	Opus (always)
Gap fixing	Opus (always)
Logic decisions, algorithmic work, state management	Opus

These rules are enforced in the orchestrator guidance, not left to discretion.

NFR-COST-02: Large-spec routing MUST tier model selection by section size

Requirement: When a specification has breakout section files, the orchestrator MUST build a structural index by measuring section file sizes and route implementation sub-agents by estimated token count. Sections below 5,000 tokens MAY be grouped two to three per agent using Sonnet. Sections between 5,000 and 20,000 tokens MUST use one Sonnet agent per section. Sections above 20,000 tokens MUST use Opus.

Rationale: Section size is a proxy for complexity and context load. Grouping small sections reduces agent dispatch overhead and cost. Large sections warrant Opus not just because they are expensive to process, but because longer sections are more likely to encode complex interactions, cross-cutting constraints, and edge cases where cheaper models produce incomplete implementations.

How the skill meets it: The sub-agent delegation strategy reference (`skills/implement/references/sub-agent-strategy.md`) specifies this routing table explicitly. The orchestrator evaluates section sizes during the planning phase and records routing decisions in the tracker.

NFR-COST-03: Sonnet agents MUST support DIGEST-based escalation to Opus

Requirement: Sonnet implementation agents MUST include a `digest` field in their structured summary output. The orchestrator MUST evaluate this digest against a complexity category table. When a match is found, Opus re-review of that agent's output is MANDATORY and non-discretionary. See §5.4.2 for the canonical complexity category table and signal keywords. See also §6.5 for how DIGEST escalation integrates with the implementation sub-agent workflow.

"Opus re-review" means the opus agent reviews the sonnet agent's code changes and findings, focusing on the complexity area identified by the DIGEST signal. See §5.4.3 for the operational procedure.

Rationale: The size-based routing in NFR-COST-02 is a heuristic. It will sometimes send moderately complex work to Sonnet that turns out to involve algorithmic subtleties, state machine interactions, or security-boundary logic. DIGEST-based escalation provides a runtime correction mechanism: the agent itself signals when it encountered something that warrants higher-model review.

How the skill meets it: Complexity categories that trigger mandatory escalation include: algorithm/calculation/formula logic, state machine transitions, permission and access control, complex conditional business rules, and cross-cutting system-wide constraints. When any of these appear in the `digest` field, the orchestrator dispatches an Opus review agent without rationalisation or override.

---

§7.3 Scalability

NFR-SCALE-01: The skill MUST remain effective for specifications with thousands of requirements

Requirement: The skill's methodology MUST not degrade in correctness or completeness as specification size increases from tens to thousands of requirements. It MUST NOT require the full specification to be loaded into any single context window.

Rationale: Real-world project specifications in active use by this skill reach several thousand requirements. This is not a theoretical limit — it is an observed usage pattern. Any design that requires the entire specification to be in context simultaneously would be fundamentally unscalable.

How the skill meets it: The per-requirement verification model (NFR-SCALE-02), structural indexing for large multi-file specs, and the section-reference anchoring system collectively ensure that no single agent, orchestrator action, or tool invocation requires loading the full specification. Each unit of work operates on the minimum specification context needed to perform that unit.

NFR-SCALE-02: Verification MUST scale linearly via per-requirement parallelism

Requirement: The verification phase MUST dispatch one sub-agent per requirement. The total verification throughput MUST scale linearly with the number of parallel agents the platform supports. For medium-sized specifications, the expected parallelism is 20 to 40 or more concurrent verification agents.

Rationale: Batching multiple requirements per verification agent was empirically found to produce less thorough results than one agent per requirement (see §1.5, Iteration 8). Single-requirement agents also scale more predictably: the total verification time is approximately the time for one agent, bounded by platform parallelism limits, rather than growing with specification size.

How the skill meets it: Each verification agent receives exactly one requirement ID, the relevant specification section text, and a pointer to the implementation under test. Agents write JSON result fragments to `.impl-verification/fragments/<req-id>.json`. The Python tool `verify_report.py` assembles these fragments deterministically after all agents complete.

NFR-SCALE-03: Structural indexing MUST be used for large multi-file specifications

Requirement: For specifications with breakout section files, the orchestrator MUST build a structural index (section IDs, file paths, estimated token sizes) before beginning implementation. This index MUST be used to route work without loading full section content into the orchestrator's context.

Rationale: Loading every section of a large specification into the orchestrator to decide which section to work on next is a self-defeating strategy: it consumes exactly the context that needs to be preserved for implementation. Structural indexing allows the orchestrator to reason about the specification's shape without ingesting its substance.

How the skill meets it: The planning phase instructs the orchestrator to run `wc -c` across section files, build a routing table, and record it in the tracker. Section content is loaded by sub-agents only when that section is the active work item.

---

§7.4 Reliability and Recovery

NFR-REL-01: The tracker MUST be self-recovering after context compaction

Requirement: The tracker file MUST contain embedded recovery instructions sufficient for the orchestrator to reconstruct sufficient session state for resumption after a context compaction event or session restart, without developer intervention.

Rationale: Context compaction is not an exceptional failure mode — it is an expected event in any sufficiently long implementation session. A skill that requires developer intervention to recover from compaction imposes an operational burden that undermines its value proposition. The tracker must make recovery automatic.

How the skill meets it: The tracker file opens with an explicit "Recovery Instructions" section as the first substantive content block. These instructions tell a freshly-loaded orchestrator, in priority order: what phase it is in, what to read to reconstruct state, and what not to do (e.g., do not begin from scratch, do not start implementation in the orchestrator directly). The tracker's structure is designed so that reading it alone is sufficient to resume work.

NFR-REL-02: The TaskList MUST survive context compaction

Requirement: Implementation task progress MUST be recorded in a mechanism that survives context compaction events and is visible to a freshly-loaded orchestrator without file I/O.

Rationale: The orchestrator's in-context task list evaporates on compaction. If task progress is only tracked in the conversation, a compaction event requires the orchestrator to reconstruct which tasks were done and which remain — a fallible process that risks re-doing completed work or skipping incomplete work.

How the skill meets it: The skill uses the Claude Code `TaskList` facility, which persists independently of conversation context and is surfaced to a new conversation automatically. The tracker recovery instructions include an explicit step to run `TaskList` to see task-level progress.

Dependency note: This requirement depends on Claude Code's TaskList facility persisting across context compaction. If TaskList does not survive compaction, the tracker file alone MUST be sufficient for recovery — the Recovery Instructions section is designed for this fallback.

NFR-REL-03: Sub-agent completion MUST use a `.done` marker polling pattern

Requirement: Sub-agent completion MUST be signalled via a `.done` marker file written by the agent when its work is finished. The orchestrator MUST use `wait_for_done.py` to poll for these markers rather than relying on the agent's conversational return.

Rationale: Conversational return signals are unreliable for coordinating parallel sub-agents — there is no guarantee that all agents have finished before the orchestrator attempts to assemble results. Marker file polling provides a deterministic synchronisation point that is independent of conversation turn ordering.

How the skill meets it: All sub-agent prompt templates include an instruction to write a `.done` file to a specified path as the final step. `wait_for_done.py` polls for the presence of all expected marker files before signalling to the orchestrator that assembly can begin.

Failure response: If a sub-agent fails to write a `.done` marker within the timeout period, the orchestrator MUST log the failure and report which agents did not complete. The orchestrator SHOULD continue with available results rather than blocking indefinitely. The orchestrator MUST NOT retry automatically — the user must be informed and can choose to re-dispatch.

NFR-REL-04: Session resume MUST include spec freshness and worktree validation

Requirement: On `/implement continue`, the orchestrator MUST verify that the specification has not changed since the tracker was last updated. For worktree-based implementations, the orchestrator MUST verify that the expected worktree exists and is on the correct branch before resuming.

Rationale: Resuming an implementation session against a modified specification without acknowledgement risks producing implementation that satisfies an outdated requirement set. Similarly, resuming in the wrong worktree can corrupt work across concurrent feature branches.

How the skill meets it: The continue phase includes an explicit spec freshness check and a worktree validation step. If either check fails, the orchestrator surfaces the discrepancy to the developer before proceeding. For single-file specs, freshness is checked by comparing the file's modification timestamp (mtime) against the `**Spec Baseline**` date recorded in the tracker (FR-5.6). For multi-file specs, freshness is checked by comparing current byte counts against the structural index stored in the tracker (§8.4.1). Hash-based change detection is not implemented.

---

§7.5 Deterministic Processing

NFR-DET-01: Report assembly MUST be performed by deterministic tools, not LLM inference

Requirement: The assembly of verification results from sub-agent output fragments into a final verification report MUST be performed by a deterministic Python script (`verify_report.py`), not by an LLM. The LLM's role is to reason over the assembled report, not to assemble it.

Rationale: LLM-based assembly of structured data introduces hallucination risk — the model may invent summary statistics, misattribute failures to requirements, or silently omit fragments it finds ambiguous. For a verification system whose purpose is to detect specification violations with high fidelity, hallucination in the assembly step would be catastrophic. Deterministic tools eliminate this risk entirely.

How the skill meets it: `verify_report.py` reads all JSON fragment files from `.impl-verification/fragments/`, validates their schema, computes pass/fail/skip counts, calculates delta from the previous run, and writes a structured report. None of these steps involve LLM inference. The orchestrator reads the assembled report and provides interpretation to the developer.

NFR-DET-02: Polling and coordination MUST use deterministic tooling

Requirement: All polling, synchronisation, and coordination operations in the verification pipeline MUST be implemented as deterministic Python scripts. LLM inference MUST NOT be used for these operations.

Rationale: The same principle that applies to assembly (NFR-DET-01) applies to coordination: LLMs should not be used to perform tasks that are better expressed as deterministic algorithms. Polling for file existence, counting completed agents, and computing deltas are all mechanical operations with no ambiguity — they are the domain of code, not inference.

How the skill meets it: `wait_for_done.py` handles all polling and completion detection. It reads the filesystem state, checks for expected marker files, and returns a deterministic status. The orchestrator invokes it as a tool and acts on its output without LLM reinterpretation of the raw filesystem state.

NFR-DET-03: The core pipeline pattern MUST be "model produces structured data → tool assembles → model reasons"

Requirement: All verification and report assembly pipelines MUST follow the model→tool→model sequence: (1) sub-agent models produce structured data written to disk; (2) deterministic tools assemble, aggregate, or transform that data; (3) the orchestrator model reasons over the assembled result. Steps 1 and 3 involve LLM

inference; step 2 MUST NOT. This pattern applies specifically to verification and report assembly — not to all orchestration steps in the skill.

Rationale: This three-step separation provides a hard boundary that prevents LLM inference from contaminating assembly operations. It also makes the pipeline auditable: the output of step 2 is a deterministic, inspectable artefact that the developer can examine independently. Any discrepancy between what sub-agents produced and what the orchestrator reported becomes traceable to a specific file rather than hidden inside a model's reasoning process.

How the skill meets it: This pattern is stated as an explicit core design principle in the skill's architecture references and is implemented end-to-end in the verification pipeline. The verification workflow is structured around this model → tool → model sequence.

---

§7.6 Security

NFR-SEC-01: The skill MUST NOT write secrets or credentials to disk artefacts

Requirement: Tracker files, verification fragments, summary JSON files, and `.done` markers MUST NOT contain API keys, passwords, tokens, or other credentials. Sub-agent task briefs MUST NOT include secrets even if the specification under implementation references secrets management. The skill operates only on specification content and code structure, not on live credentials.

Rationale: Artefacts written by the skill (trackers, fragments, summaries) are git-committed alongside implementation code by many users. A tracker or fragment that inadvertently captures a secret from a specification would expose that secret in version control history.

How the skill meets it: Sub-agent prompt templates are scoped to specification content and code artefacts. They do not request or surface runtime environment values. Users are responsible for ensuring their specifications do not embed live credentials — the skill does not validate this, but it does not propagate such content into its own artefact schema.

§8 Edge Cases & Error Handling

This section documents the edge cases and error scenarios that the `/implement` skill encounters in practice. Every scenario described here has been observed during real usage — these are not theoretical failure modes. For each scenario, the section describes the conditions under which it occurs, how the skill detects it, and the recovery or mitigation path.

MoSCoW language is used throughout: MUST indicates mandatory behaviour, SHOULD indicates strong recommendation, COULD indicates optional enhancement, and WILL NOT indicates explicit exclusion.

---

§8.1 Context Compaction Recovery

Context compaction is the most common failure mode in extended implementation sessions. When the context window fills and Claude compacts the conversation, the orchestrator loses detailed understanding of the current implementation state — specific section references, task progress, and phase awareness degrade or disappear entirely.

§8.1.1 Detection

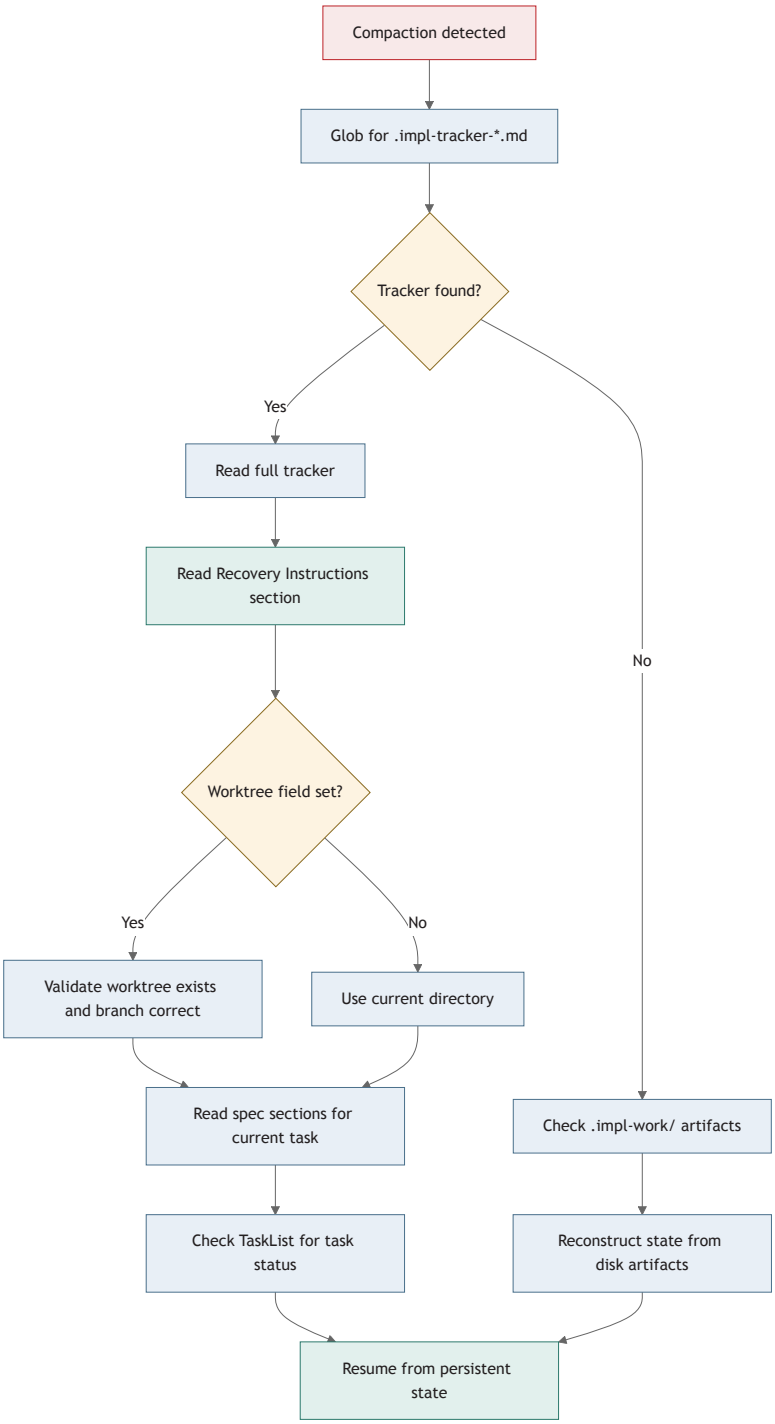
The orchestrator MUST self-monitor for the following signs of context compaction:

- A vague sense of "building something" without specific knowledge of which requirements or spec sections are being implemented
- Loss of phase awareness — uncertainty about whether the session is in Phase 2 (Implementation) or Phase 3 (Verification)
- An urge to implement directly without reading the tracker or spec sections first
- Inability to recall which sub-agents have been dispatched or what their status is
- Referring to requirements in vague prose rather than compact section references (e.g., "the authentication requirement" instead of "§4.2")

Distinguishing compaction from drift: Compaction and drift share some surface symptoms but require different responses. Compaction presents as *global* context loss — the orchestrator cannot recall any spec details, is uncertain which project it is working on, or has lost phase awareness entirely. Drift presents as *selective* context loss — the orchestrator still remembers the project and its overall state but refers to requirements in vague prose rather than section references, or begins implementing features that are not in the spec. When the loss is global, the recovery path is §8.1.2 (read tracker and spec to recover full context). When the loss is selective, the recovery path is §8.8.3 (re-read the specific spec section for the current task).

§8.1.2 Recovery Flow

Upon detecting compaction, the orchestrator MUST follow this recovery sequence:



The recovery sequence in detail:

1. **Find the tracker**: The orchestrator MUST run `Glob(".impl-tracker-*.md")` in the current directory and any known worktree paths. The tracker is the primary recovery artifact.
2. **Read the tracker**: The tracker's Recovery Instructions section MUST contain sufficient information to resume without any external context. This includes the spec path, current phase, worktree location, and the requirements matrix.

3. **Validate the worktree**: If the tracker's `**Worktree**` field is set, the orchestrator MUST verify the worktree path exists and is on the expected branch before proceeding.
4. **Re-read spec sections**: The orchestrator MUST read the spec sections relevant to the current task. Section references in the tracker survive compaction where prose descriptions do not — this is by design (§2.2, Principle 8).
5. **Check TaskList**: Task objects persist across compaction events. The orchestrator SHOULD check the task list to determine which tasks are complete, in progress, or pending.
6. **Resume**: With tracker state, spec sections, and task status re-loaded, the orchestrator MUST resume work from the point captured in persistent state, not from memory.

§8.1.3 Design Rationale

The three-layer context preservation architecture (§2.3) exists specifically to make compaction recovery possible. The tracker provides phase and requirement state. Sub-agent output files on disk provide implementation artifacts. Section references provide compact, compaction-resistant pointers to full spec content. Any single layer failing still allows recovery through the remaining two.

---

§8.2 Context Exhaustion

Context exhaustion occurs when the context window is so full that `/compact` itself cannot succeed — the conversation is unrecoverable in its current session.

§8.2.1 Detection

The orchestrator SHOULD monitor for these warning signs before exhaustion occurs:

- Responses becoming noticeably shorter or less detailed
- Tool calls failing due to output length constraints
- Sub-agent dispatches producing truncated task briefs
- More than two heavy sub-agents running in parallel (a known risk factor; see §8.2.2 for the definition of "heavy")

§8.2.2 Prevention

The skill SHOULD employ these preventive measures:

- **Parallel agent limit**: The orchestrator SHOULD NOT dispatch more than two heavy sub-agents simultaneously. A **heavy** sub-agent is one using the sonnet or opus model tier. Haiku-tier agents are **lightweight** and do not count against this concurrent agent cap (NFR-CTX-03). Each heavy dispatch consumes significant context for the task brief and the completion notification.
- **Load-on-demand discipline**: Reference files and spec sections MUST be loaded only when needed for the current phase, never preemptively (§2.5).
- **Structured output to disk**: Sub-agents MUST write their results as structured JSON to disk files, not as conversational responses. This prevents sub-agent output from inflating the orchestrator's context.
- **Context budget awareness**: Before dispatching a large batch of verification agents, the orchestrator SHOULD assess whether sufficient context remains for the assembly and reporting steps that follow.

§8.2.3 Recovery

When exhaustion does occur, the session is unrecoverable — but the implementation state is not lost:

1. All sub-agent output is safe on disk in `.impl-work/` and `.impl-verification/` directories, because sub-agents write to files rather than returning conversational output.
2. The tracker file on disk reflects the last successful state update.
3. Task objects persist independently of the conversation.

Recovery MUST proceed by starting a fresh session with `/implement continue <name>`. The Phase 5 workflow (§3.5) handles this case: it reads the tracker, validates the worktree, checks spec freshness, and resumes from persistent state.

The developer SHOULD NOT attempt to recover within the exhausted session. The correct action is to start a new conversation.

---

§8.3 Sub-Agent Failure Modes

Sub-agents can fail in several ways. The skill does not implement an explicit retry mechanism — empirically, Claude tends to self-correct on subsequent attempts. However, the orchestrator MUST detect failures and respond appropriately.

§8.3.1 Sub-Agent Produces No Output

Scenario: A sub-agent is dispatched but never writes its output file or `.done` marker to disk.

Detection: The `wait_for_done.py` tool polls for `.done` markers with a configurable timeout. The default timeout is `max(600, N × 30)` seconds, where N is the number of agents being waited on (per §6.3). If a marker does not appear within the timeout window, the tool reports a timeout.

Recovery: The orchestrator MUST treat a timeout as a failed dispatch. The orchestrator SHOULD re-dispatch the same task to a fresh sub-agent. If the second dispatch also times out, the orchestrator MUST report the failure to the developer and mark the task as `blocked` in the tracker. A `blocked` task remains in the tracker until the developer explicitly requests re-dispatch or marks it as `n/a` — there is no automatic unblock path.

§8.3.2 Sub-Agent Writes Partial Output

Scenario: A sub-agent writes an output file but the content is incomplete — truncated JSON, missing required fields, or incomplete analysis.

Detection: The orchestrator reads the sub-agent's output file after the `.done` marker appears. For verification fragments, `verify_report.py` validates JSON structure via `verification_schema.py`. Malformed fragments produce validation errors during report assembly.

Recovery: The orchestrator SHOULD re-dispatch the task. If partial output contains useful information (e.g., a correct verdict with a missing justification), the orchestrator COULD use the partial result and dispatch a supplementary agent to fill the gap. The orchestrator MUST NOT silently accept malformed output.

§8.3.3 Malformed JSON Fragments

Scenario: During verification, a sub-agent writes a JSON fragment that fails schema validation.

Detection: `verify_report.py` validates each fragment against the expected schema. Invalid fragments produce hard errors during assembly.

Recovery: Hard validation errors MUST abort report assembly. The orchestrator MUST re-dispatch the verification for the affected requirement(s). The orchestrator MUST NOT attempt to manually fix malformed JSON — the correct response is to re-dispatch with a fresh sub-agent that has a clean context.

§8.3.4 Sub-Agent Implements Incorrectly

Scenario: A sub-agent completes its task and writes output, but the implementation is incorrect — tests fail, code does not compile, or the implementation does not match the spec.

Detection: The post-task checklist (§3.2) mandates running tests after every sub-agent completion. Test failures are the primary detection mechanism. Additionally, the DIGEST escalation pattern flags complex tasks completed by sonnet-class agents for mandatory opus review.

Recovery: The orchestrator MUST NOT mark the task as complete if tests fail. The orchestrator SHOULD dispatch a fix sub-agent (always using opus for verification fixes) with the test failure output, the spec quote, and the current code state. The orchestrator MUST NOT silently modify tests to match a wrong implementation.

---

§8.4 Specification Evolution

Specifications can change between implementation sessions — requirements added, removed, or modified after implementation has already begun.

§8.4.1 Detection

Phase 5 (Continue) MUST perform a spec freshness check when resuming work. The detection mechanisms differ by spec type:

Multi-file specs: The orchestrator MUST compare current file sizes (`wc -c`) against the structural index stored in the tracker. The following changes are flagged:

Change Type	Detection Rule
New section files	Files present on disk but absent from the structural index
Removed section files	Files listed in the structural index but absent from disk
Modified sections	Byte count changed by more than 20% from the stored value
Sub-split patterns	A previously single file now has letter-suffix variants (e.g., `02a-`, `02b-`, `02c-`)

Single-file specs: The orchestrator MUST compare the file's modification timestamp (mtime) against the tracker's `**Spec Baseline**` date. A newer mtime triggers the freshness warning. Note that mtime reflects the last write to the file, not necessarily a content change — a `touch` command or file copy will trigger a false positive. The orchestrator SHOULD inform the developer of this limitation when the freshness warning fires.

STRUCT check: The orchestrator SHOULD check for `.spec-tracker-*.md` files containing a `## Pending Structural Changes` section, which indicates the specification is mid-restructure.

§8.4.2 Resolution Options

When specification changes are detected, the orchestrator MUST present the developer with three options:

1. **Re-scan affected sections**: Re-read only the changed files, extract new/modified/removed requirements, update the requirements matrix, create tasks for new requirements, and mark removed requirements as `n/a` with a note such as "requirement removed in spec update — [date]". Orphaned tasks — tasks in the tracker whose corresponding requirements no longer appear in the spec — MUST also be marked `n/a` with the same note. This is appropriate when changes are localised.
2. **Proceed as-is**: Acknowledge the changes without re-scanning. The orchestrator MUST log the detected changes in the Implementation Log with specific details (file names, size deltas). This is appropriate when the developer knows the changes do not affect in-progress work.
3. **Full re-plan**: Archive the current tracker with a date suffix, then re-run Phase 1 from scratch. Completed work, known gaps, and log history MUST be carried forward from the archived tracker using the following field-by-field rules. This is appropriate when changes are extensive or structural.

Tracker Field	Carry-Forward Rule
Requirements Matrix	Requirements with `complete` status retain their status and their Implementation/Tests references. Requirements that match changed spec sections reset to `pending`. New requirements discovered during re-scan are added as `pending`. Requirements whose spec sections were removed are marked `n/a` with note "removed in spec update."
Implementation Log	All previous entries are preserved in full, preceded by a separator line noting the re-plan date and reason. Log entries are NOT summarised — the full history is kept.
Known Gaps	Carried forward as-is. Gaps that are no longer relevant because the relevant spec section changed are marked "resolved by spec update" with the date.
Structural Index	Rebuilt from scratch using current `wc -c` values from the updated spec. Previous values are discarded.
Deviations	Carried forward as-is. The developer SHOULD review each deviation for continued relevance given the spec changes.

The orchestrator MUST NOT silently proceed when specification changes are detected. The developer MUST be informed and MUST choose a resolution path.

---

§8.5 Worktree Edge Cases

Git worktree support enables concurrent, isolated implementations. Several edge cases arise around worktree lifecycle.

§8.5.1 Worktree Deleted Between Sessions

Scenario: The tracker references a worktree path that no longer exists on disk.

Detection: Phase 5 validation MUST verify that the worktree path exists and appears in `git worktree list` output.

Recovery: The orchestrator MUST warn the developer that the worktree has been removed. The orchestrator MUST present three options:

- Re-create the worktree at the same path and branch
- Switch to working in the current directory instead
- Abort and let the developer resolve manually

The orchestrator MUST NOT silently fall back to the current directory — the developer needs to know that the isolation boundary has changed.

§8.5.2 Wrong Branch

Scenario: The worktree exists but is on a different branch than the tracker's `**Branch**` field specifies.

Detection: Phase 5 validation MUST check the current branch of the worktree against the expected branch stored in the tracker.

Recovery: The orchestrator MUST warn the developer about the branch mismatch. The developer decides whether to switch the worktree to the expected branch, update the tracker to reflect the current branch, or abort.

§8.5.3 Git Not Available

Scenario: The tracker contains a worktree path but `git` is not available in the current environment (e.g., a container without git installed).

Detection: The orchestrator SHOULD check for git availability before attempting worktree validation.

Recovery: The orchestrator MUST fall back to the current directory with a warning explaining that worktree validation was skipped because git is unavailable. Work COULD proceed but the developer SHOULD be aware that branch isolation is not being enforced.

---

§8.6 Concurrent Session Handling

Two simultaneous sessions attempting to work on the same implementation produce undefined behaviour. The skill does not implement file locking or coordination between sessions.

§8.6.1 The Problem

If two sessions edit the same tracker file concurrently, one session's updates will overwrite the other's. There is no merge strategy — the last writer wins, and the other session's state becomes inconsistent with the tracker on disk.

Similarly, two sessions implementing different tasks from the same tracker could produce conflicting code changes (e.g., both modifying the same file).

§8.6.2 Prevention via Worktrees

Git worktree support was added specifically to address concurrent work. Each worktree provides:

- An isolated file system working tree (no conflicting code edits)
- Its own tracker file (no shared tracker state)
- Its own `.impl-work/` and `.impl-verification/` directories

The orchestrator SHOULD recommend worktree isolation when the developer is working on multiple features from the same codebase. Each concurrent implementation SHOULD operate in its own worktree with its own tracker.

§8.6.3 What the Skill Will Not Do

The skill WILL NOT implement:

- File locking on tracker files

- Session coordination or handoff protocols
- Conflict detection between concurrent sessions
- Automatic merge of concurrent tracker updates

These are outside the skill's scope. The worktree mechanism provides sufficient isolation for the expected use case (one developer, multiple features). Multi-developer coordination is a project management concern, not an implementation skill concern.

---

§8.7 Stale Notifications

When resuming via `/implement continue`, delayed completion notifications from a previous session's sub-agents may arrive in the new session.

§8.7.1 Scenario

A previous session dispatched sub-agents with `run_in_background: true`. The session ended (or the context was exhausted) before all agents completed. When a new session starts and the orchestrator begins dispatching new work, completion notifications from the old agents arrive unprompted.

§8.7.2 Detection

The orchestrator MUST apply this rule: **if you did not dispatch the agent in the current conversation, the notification is stale.**

Stale notifications are identifiable because:

- The orchestrator has no record of dispatching the agent in its current conversation history
- The agent's task may have already been completed or superseded by the current session's work
- The output files on disk may already reflect the agent's work (it wrote to disk before the old session ended)

§8.7.3 Handling

The orchestrator MUST silently ignore stale notifications. Specifically:

- The orchestrator MUST NOT read TaskOutput for agents it did not dispatch
- The orchestrator MUST NOT update the tracker based on stale notifications
- The orchestrator MUST NOT treat stale completions as progress on current-session work
- The orchestrator SHOULD continue with its current task without interruption

The sub-agent's actual output — the files it wrote to disk — is already available and was either picked up by the previous session or will be naturally discovered by the current session when it reaches that task.

---

§8.8 Implementation Drift Detection

Implementation drift occurs when the orchestrator or a sub-agent diverges from the specification during implementation — building something different from what the spec requires, making assumptions instead of reading the spec, or skipping requirements.

§8.8.1 Self-Evaluation Checklist

The orchestrator MUST periodically evaluate itself against the following checklist. Any checked item indicates active drift that requires immediate correction:

- ☐ Implementing something not mentioned in the specification
- ☐ Making assumptions about behaviour instead of reading the spec
- ☐ Using "I think" or "probably" instead of "the spec says" or "§N.M requires"
- ☐ Skipping a requirement because it seems hard or complex
- ☐ Combining multiple features or requirements into a single task when the plan separated them
- ☐ Modifying tests to match an incorrect implementation instead of fixing the implementation
- ☐ Proceeding without reading the relevant spec section before starting a task
- ☐ Marking a task as complete without running tests
- ☐ Referring to requirements in vague prose rather than by section reference
- ☐ Adding features or behaviours that the spec does not mention ("gold plating")

§8.8.2 When to Self-Evaluate

The orchestrator MUST run the self-evaluation checklist:

- Before marking any task as complete
- After context compaction recovery (§8.1)
- When the developer questions whether the implementation matches the spec
- When the orchestrator catches itself using hedging language ("I think", "probably", "should be")

The orchestrator SHOULD also self-evaluate:

- At the start of each new task
- When switching between spec sections
- After a sub-agent returns with unexpected results

§8.8.3 Correction

When drift is detected:

1. The orchestrator MUST stop current work immediately
2. The orchestrator MUST re-read the relevant spec section(s) — the actual text, not a memory of what it said
3. The orchestrator MUST assess whether work already completed is affected by the drift
4. If affected work exists, the orchestrator MUST either fix it or document the deviation in the tracker with the developer's consent
5. The orchestrator MUST NOT continue implementing until the drift is resolved

§8.8.4 Sub-Agent Drift

Sub-agents are structurally resistant to drift because they start with fresh context windows containing only the relevant spec sections (§2.2, Principle 3). However, drift can still occur if:

- The task brief passed to the sub-agent is inaccurate or incomplete
- The sub-agent interprets an ambiguous spec requirement differently from the developer's intent
- The sub-agent makes assumptions about surrounding code that are incorrect

Detection relies on the post-task checklist: test execution, DIGEST escalation for sonnet agents, and optional spec compliance checks. The orchestrator MUST NOT assume sub-agent output is correct without running these checks.

---

§8.9 Test Integrity Edge Cases

Tests serve as the primary mechanical check that implementation matches specification. Several edge cases threaten test integrity.

§8.9.1 Tests Pass Unexpectedly (TDD Mode)

Scenario: In TDD mode, the test-writing agent produces tests that pass before any implementation work is done. This indicates either the feature already exists, or the tests are too loose to detect the absence of the feature.

Detection: TDD mode MUST run the newly written tests before implementation begins. If tests pass at this stage, the orchestrator MUST flag the situation.

Recovery: The orchestrator MUST NOT proceed with implementation without resolving this. Options:

1. **Feature already exists**: The orchestrator SHOULD verify that the existing implementation actually satisfies the spec requirement (not just that the tests pass). If it does, mark the requirement as `complete` with a note.
2. **Tests too loose**: The orchestrator MUST dispatch a new test-writing agent to produce more specific tests. The new tests MUST fail before implementation begins.
3. **Ambiguous**: Present the situation to the developer for resolution.

§8.9.2 Tests Fail for Setup Reasons

Scenario: Tests fail not because the implementation is wrong, but because of import errors, missing fixtures, incorrect paths, missing dependencies, or environment configuration issues.

Detection: The nature of test failures reveals the cause. Import errors, `ModuleNotFoundError`, `FileNotFoundError`, and fixture-related exceptions indicate setup problems rather than spec violations.

Recovery: The orchestrator MUST fix the test setup without modifying test assertions or expected values. The distinction is critical:

- **Fix**: Import paths, fixture configuration, test data paths, dependency installation
- **Do not fix**: Assertion values, expected outputs, test logic, requirement coverage

If it is unclear whether a failure is a setup issue or a genuine spec violation, the orchestrator SHOULD treat it as a spec violation and fix the implementation, not the test.

§8.9.3 Sub-Agent Modifies Tests to Match Wrong Implementation

Scenario: A sub-agent, when faced with test failures, modifies the tests to match its implementation rather than fixing the implementation to match the tests.

Detection: This is difficult to detect automatically. The post-task checklist, DIGEST escalation, and spec compliance checks provide partial coverage. Verification (Phase 3) provides the definitive check — verification sub-agents re-read the spec independently and will flag implementations that do not match.

Mitigation: The skill enforces a hard rule: sub-agents MUST NEVER modify test assertions to make a failing implementation pass. This rule MUST be included in sub-agent task briefs. Specifically:

- Implementation sub-agents MUST NOT edit test files
- TDD implementation sub-agents receive the test file path for reference but MUST NOT modify it
- Fix sub-agents MUST NOT alter test expectations unless explicitly instructed by the developer

If a sub-agent is found to have modified tests, the orchestrator MUST revert the test changes and re-dispatch the implementation task. Revert is performed via `git checkout -- <test-file-path>` on the affected test files to restore

them to their pre-modification state. If the test files were newly written by the test-writing agent and have no prior git state to restore, the orchestrator MUST re-dispatch the test-writing agent with the original spec requirements to regenerate clean tests before re-dispatching the implementation task.

§8.9.4 Test-Implementation Circular Dependency

Scenario: Tests require infrastructure (database, API, file system state) that only exists after implementation, but the TDD workflow requires tests to exist before implementation.

Detection: Test failures during the initial TDD "run tests to confirm they fail" step may reveal that tests cannot even execute without the implementation infrastructure.

Recovery: The orchestrator SHOULD structure tasks so that infrastructure setup is a separate, non-TDD task completed before TDD tasks that depend on it. When this is not possible, the test-writing agent SHOULD use appropriate mocking or fixtures to isolate the test from infrastructure dependencies. The orchestrator MUST NOT abandon TDD mode for the entire implementation because of infrastructure dependencies in a few tasks — the correct response is targeted task reordering.

§9 Constraints, Assumptions & Out of Scope

§9.1 Platform Constraints

The `/implement` skill is tightly coupled to its execution environment and cannot degrade gracefully if certain dependencies are unavailable.

Hard Dependencies

- **Claude Code CLI**: Requires the full Claude Code platform. The skill uses platform-specific features that have no fallback:
  - Task tool for spawning background agents
  - TaskCreate, TaskUpdate, TaskList, TaskGet for persistence across context compaction
  - Sub-agent spawning and parallel task delegation
  - Background task execution (`run_in_background` parameter)
- **Model Tier Availability**: Hard dependency on three Claude model tiers being available during a single implementation session (see NFR-COST-01 and NFR-COST-02 for the routing rules that govern when each tier is selected):
  - **Haiku** (fast, cost-efficient): Boilerplate tasks, simple spec parsing, routine delegation
  - **Sonnet** (balanced): Standard implementation tasks, moderate complexity work
  - **Opus** (most capable): Verification (always), gap fixing (always), complex implementation, planning for large specs

  No automatic fallback: Tier selection is quality-critical — especially for verification, where Opus is required. The skill does NOT automatically fall back to a lower tier if the selected tier is unavailable. Instead, the skill informs the user with an actionable message. For example, if Opus is unavailable when verification is attempted:

  > *"Opus is required for verification but is currently unavailable. Please try again when Opus access is restored, or explicitly request Sonnet-based verification if you accept reduced accuracy."*

  The user must explicitly choose to proceed with a lower tier. Automatic silent fallback is a deliberate non-feature: silently downgrading verification quality could produce misleading compliance results.

Soft Dependencies (Recommended, Not Required)

- **Python 3**: Used by two verification tools:
  - `tools/verify_report.py` — assembles JSON fragments into a deterministic markdown report
  - `tools/wait_for_done.py` — polls for sub-agent completion markers

  Python 3 is a **soft dependency** — assumed available in the environment (standard on macOS and Linux). If Python 3 is absent, these tools cannot run; the skill falls back to orchestrator-managed alternatives: manual fragment assembly and orchestrator-driven polling loops. See §9.3 for the consistent framing of this assumption.
- **Git**: Strongly recommended for version control of trackers and implementation state, but not required. Worktree support assumes git is available; without it, concurrent sessions require manual directory management.

---

§9.2 Design Constraints

Context Window is the Fundamental Constraint

Everything in the skill's architecture — from tracker format to verification pipeline — is shaped by context window limitations and Claude's compaction behavior.

- **Persistent state via markdown files, not conversation memory**: Trackers, verification reports, and preferences live on disk because conversation context is transient. Context compaction loses conversation history; disk state survives.
- **Section references as stable anchors**: Specs are strongly recommended to use numbered section references (§2.1, §3.4) or unambiguous prose headings because these survive context compaction; conversational references ("the part about authentication") do not. Section references are not required — the skill degrades gracefully without them (verification produces coarser-grained results, and V-item matching uses text similarity as a fallback) — but structured §N.M references produce the most reliable outcomes.
- **Structured output over conversational answers**: Sub-agents write JSON fragments to disk, not conversational summaries (NFR-DET-03: "model produces structured data → tool assembles deterministically → model reasons over results"). This is intentional and necessary.
- **Skill restructuring for efficiency**: SKILL.md itself was reduced from ~1400 to ~296 lines in February 2026 specifically to manage context overhead. Large prompt files are broken into reference materials loaded on-demand per phase.

File-Based State Interchange

The skill uses markdown and JSON files as its primary data interchange format:

- **Trackers**: Markdown with structured field markers (`**Field**:`) and matrix tables
- **Verification fragments**: JSON with requirement ID, status, findings, evidence
- **Preferences**: Markdown with simple key-value pairs

This choice prioritizes human readability and git-friendly diffs over binary serialization.

Markdown as Spec Format

The skill works with markdown specifications. It does *not* parse:

- YAML spec files
- JSON schemas
- Confluence/Jira exports
- Docx, PDF, or proprietary formats

If source specifications exist in other formats, they must be converted to markdown first (that is outside this skill's scope — the `/spec` skill is responsible for initial spec creation).

No Explicit Retry Logic for Sub-Agent Failures

When a sub-agent (verification agent, implementation agent, test writer) fails or returns unexpected output:

- There is no built-in retry mechanism
- The skill relies on Claude's natural tendency to self-correct in follow-up attempts
- Users can manually re-run verification or re-assign implementation tasks

This is a practical trade-off to keep skill complexity low; explicit retry logic adds significant overhead.

---

§9.3 Assumptions

Specification Pre-Requisites

- **Specifications are pre-written**: The `/implement` skill *consumes* finished specifications; it does not write them. The `/spec` skill is responsible for specification authoring. Users are assumed to arrive with a completed (or near-complete) spec document.
- **Specifications have discrete, testable requirements**: Specs work best when they articulate individual requirements as "must," "should," or "could" statements rather than prose paragraphs. The skill's verification phase depends on being able to extract and verify discrete requirements.
- **Section references are helpful but not mandatory**: Specs with numbered sections (§1.2, §3.4) and unambiguous prose headings work best. The skill degrades gracefully without them—section references are strong anchors, but the skill can still parse and verify unstructured prose, with reduced precision.

User Environment

- **Python 3 is assumed available** (soft dependency — standard on macOS and Linux): The deterministic verification tools (`tools/verify_report.py`, `tools/wait_for_done.py`) require Python 3. If Python 3 is not present, the skill falls back to orchestrator-managed alternatives (manual fragment assembly, orchestrator-driven polling loops) — see §9.1 for the full degradation behaviour.
- **Git is available** (or user manages concurrent sessions manually): Worktree support was added in February 2026 to handle concurrent sessions across multiple branches. Without git, users must manually segregate working directories and update tracker paths. See §8.5.3 for the skill's behaviour when git is unavailable at runtime.
- **Users understand implementation workflows**: Users are assumed to be familiar with either TDD or traditional implementation patterns, and can meaningfully review and validate tests that verify spec requirements. Note: the TDD sub-agent (§3.2.3) writes the tests from scratch — this assumption is about the user's ability to evaluate whether the generated tests correctly represent the spec intent, not about the user writing tests themselves.

Skill Behavior Assumptions

- **Users will read the tracker**: The tracker is the source of truth during recovery after context compaction. The skill assumes users will read it and follow its recovery instructions rather than relying on conversational cues.
- **Tests are run after every implementation**: The skill's verification pipeline assumes that tests have been run and pass. Implementation without passing tests is treated as incomplete.
- **Users will approve plans before implementation**: Phase 1 produces a plan (tasks, task order, scope summary) that MUST be approved before Phase 2 begins. The skill assumes this gate is not bypassed.

---

§9.4 Out of Scope

Specification Authoring

- **No spec writing**: The `/implement` skill does not write or author specifications. It takes finished specifications as input. If your spec needs to be written or significantly revised, use the `/spec` skill first.

Deployment & Infrastructure

- **No code deployment**: The skill stops at "implementation complete with tests passing." It does not:
  - Push code to production
  - Deploy infrastructure
  - Run deployment pipelines
  - Update live services
- **No CI/CD pipeline integration**: The skill cannot:
  - Trigger Jenkins, GitHub Actions, or other CI systems
  - Hook into automated test runners
  - Consume pipeline outputs or feedback loops
  - Push code to remote repositories (users can do this manually after verification)
- **No infrastructure-as-code generation**: Infrastructure tooling (Terraform, CloudFormation, Kubernetes manifests) is out of scope. The skill focuses on application code implementation.

Project Management Integration

- **No Jira, GitHub Issues, or other PM tool integration**: The skill does not:
  - Create or update tickets
  - Sync with project management systems
  - Consume requirements from PM tools
  - Report progress back to PM systems
- **No team workflow management**: No Slack notifications, no calendar blocking, no standup generation.

Code Quality & Non-Functional Aspects

- **Limited quality checks**: The skill performs spec-compliance verification but does *not* include:
  - Deep code reviews (style, architecture, design patterns)
  - Security vulnerability scanning
  - Performance profiling or optimization
  - Accessibility compliance checks
  - Documentation generation beyond what specs require

  Rationale: The primary focus is requirement-to-implementation fidelity, not code quality. Code quality should be managed by separate linting, testing, and review workflows outside this skill.
- **No refactoring**: While the skill *can* be used for refactoring work (and has been in practice), it is not optimized for it. Refactoring typically requires a different workflow (iterative improvement with optional requirements) rather than the requirement-driven model this skill assumes.

Concurrent Version Management

- **Limited concurrent session support**: The skill has basic worktree support (added Feb 2026) but does not manage:
  - Automatic merge of concurrent changes
  - Conflict resolution
  - Branch synchronization

  Users are responsible for git branching strategy and merging parallel work.

Web Version Maintenance

- **claude-web/ is not actively maintained**: A web adaptation exists in the repository but is stale and not part of the active skill roadmap. The CLI version (skills/implement/) is canonical.

---

§9.5 Future Considerations

Claude Code Version Resilience

Current state: No hard version constraints on Claude Code CLI; the skill uses platform features (Task tool, TaskCreate/Update, background agents) that could change in future versions.

Future: Consider pinning minimum Claude Code versions if the platform makes breaking changes to Task tool semantics or sub-agent spawning.

CI/CD Hook Points

Current state: No pipeline integration.

Future: Could add optional integration points:

- Consume spec changes from a GitHub branch
- Push verified implementation to a PR
- Trigger verification on code review
- Report verification results back to GitHub

This would require careful design to avoid over-coupling the skill to CI/CD services.

Web Version Revival

Current state: `claude-web/` directory exists but is not actively maintained. The web-based Claude interface does not support Task tool or sub-agent spawning, making full skill replication impossible.

Future: If web Claude gains Task tool support, could revive and align the web version with the CLI version.

Quality Gate Extensibility

Current state: Limited quality checks; focus is spec compliance.

Future: Could add optional gate for:

- Linting checks (Python flake8, JavaScript ESLint, etc.)
- Type checking (mypy, TypeScript, etc.)
- Basic security scans
- Test coverage thresholds

These would be opt-in to keep the skill focused; users with strict quality requirements can layer them on top.

Specification Evolution During Implementation

Current state: Basic spec freshness detection exists (§8.4 edge cases); no full merge/diff logic.

Future: Could add:

- Automatic spec-to-tracker reconciliation when specs change mid-implementation

- Diff-based notification of requirement changes
- Interactive conflict resolution (old requirement changed? remove, add new, or modify implementation?)

Parallel Verification Optimization

Current state: Parallel sub-agents work well at scale (documented scale: 20-40+ agents, several thousand requirements).

Future: Could optimize further with:

- Automatic batching of small requirements (group 5-10 small reqs into one agent) — **note**: this would require relaxing FR-3.15, which currently mandates one verification agent per requirement. Any batching proposal must explicitly amend that rule rather than working around it.
- Smarter model tier routing (very simple reqs → Haiku always, complex edge cases → Opus)
- Caching of frequently-verified requirements (edge cases, common patterns)

Summary

The `/implement` skill is fundamentally constrained by context windows, model tier availability, and the need for persistent disk-based state. It assumes specifications are pre-written, Python 3 and git are available (soft dependencies with graceful degradation if absent), and users will follow the planning → implementation → verification workflow. Model tier selection is quality-critical — there is no silent automatic fallback; if a tier is unavailable, the skill informs the user and requires an explicit choice to proceed at reduced accuracy. Section references are strongly recommended but not required; the skill degrades gracefully without them. It deliberately excludes spec authoring, deployment, CI/CD, and deep code quality work. Future evolution should focus on version resilience, optional CI/CD hooks, and quality gate extensibility while maintaining the skill's core identity as a specification-fidelity tool.