

Atelier: The Agentic Operating System for Next-Generation Software Engineering

Executive Summary: The Architecture of Agentic Orchestration

The paradigm of software engineering is currently navigating a pivotal phase transition, evolving from a craft centered on the manual authoring of syntax to a discipline of systemic orchestration. This evolution is precipitated by the maturation of reasoning-capable Large Language Models (LLMs) and the advent of agentic Command Line Interfaces (CLIs) such as Claude Code. In this emerging epoch, the primary unit of value contribution for the senior engineer shifts from the function or the class to the **specification**, the **architectural constraint**, and the **verification protocol**. The developer is no longer merely a writer of code but becomes an architect of systems, orchestrating fleets of autonomous agents that execute the labor of implementation.

This report presents a comprehensive architectural blueprint for **Atelier**, a proposed tool designed to act as an Operating System for this new agentic workflow. Atelier wraps the capabilities of the Claude Code CLI into a rigorous, structured pipeline: **Idea** \rightarrow **Gather** \rightarrow **Specify** \rightarrow **Design** \rightarrow **Plan** \rightarrow **Build** \rightarrow **Review** \rightarrow **Deploy**. By leveraging the design principle of **Progressive Disclosure** and the operational power of **Multi-Agent Teams**, Atelier operationalizes the shift from "vibe coding" to **Agentic Engineering**. It treats Product Requirements Documents (PRDs), Technical Design Documents (TDDs), and Execution Plans not as passive documentation, but as the executable source code of the agentic runtime. The following analysis provides an exhaustive examination of the theoretical foundations, technical implementation, and operational workflows of Atelier. It dissects the necessary strategies for local-first state management, defines precise artifact schemas for agent consumption, and establishes the control plane mechanisms required to construct a robust, self-healing software factory. We explore how deterministic hooks, shared-file state machines, and hierarchical agent topologies converge to create a development environment where human intent is rigorously translated into verifiable, production-grade software.¹

Part I: The Theoretical Foundation of Agentic Engineering

1.1 The Collapse of "Vibe Coding" and the Rise of Rigor

The initial integration of LLMs into software development was characterized by a phenomenon widely termed "vibe coding"—an ad-hoc, conversational methodology where developers prompted models to generate code snippets based on loose, unstructured intent. While this approach proved effective for rapid prototyping and overcoming the "blank page" problem, it has fundamentally failed to scale to the demands of enterprise engineering. "Vibe coding" suffers from critical structural weaknesses: **context drift**, where the model loses track of requirements over long conversational turns; **lack of reproducibility**, where the same prompt yields vastly different outputs; and the accumulation of "**hallucinated technical debt**," a patchwork of unverified logic that ostensibly functions but fails under edge cases or integration stress.⁴

Agentic Engineering, the philosophy that underpins the Atelier architecture, explicitly rejects the stochastic nature of vibe coding in favor of structured, deterministic workflows. It posits that an AI agent should not be treated as a chatbot, but rather as a probabilistic compute engine that requires precise inputs (context), strict constraints (guardrails), and verifiable outputs (tests). In this model, the ephemeral "chat" is replaced by the persistent **Artifact**—a version-controlled document that serves as the single source of truth for the agent's actions. The transition to Agentic Engineering necessitates a fundamental "Shift Left" in cognitive load. The human developer must expend significant effort upfront in the **Specify** and **Design** phases, crafting rigorous PRDs and TDDs that leave little room for ambiguity. The "Build" phase, traditionally the most time-consuming aspect of software development, is inverted into an automated execution step performed by agents. This inversion aligns with methodologies like SPARC (Specification, Pseudocode, Architecture, Refinement, Completion), which provide a structured scaffolding for the interaction between the human architect and the AI builder.⁶

1.2 The Operating System Metaphor

To effectively manage the complexity of multi-agent workflows, Atelier is conceptualized not merely as a plugin or a script, but as a dedicated **Operating System (OS)** for development. Just as a traditional kernel manages hardware resources (CPU cycles, RAM, I/O) to execute diverse processes, Atelier manages **Context**—the scarcest and most critical resource in the LLM era—to execute **Workflows**.

- **The Kernel:** The claude code CLI functions as the kernel of this OS. It provides the low-level system calls, granting access to the file system, shell execution environments, and the reasoning capabilities of the underlying model (e.g., Claude 3.5 Sonnet or Opus). It handles the "context switching" between different tasks and manages the token budget.⁸
- **User Space:** Atelier constitutes the user space, providing the shell (Terminal User Interface), the process manager (State Machine), and the memory management (Artifact Storage) required to run complex applications.
- **Drivers:** The Model Context Protocol (MCP) servers act as device drivers. They allow the kernel to interface with external peripherals and data sources—such as GitHub

repositories, Jira boards, PostgreSQL databases, or web browsers—standardizing the I/O between the agent and the outside world.¹⁰

- **File System:** The "memory" of this OS is not volatile RAM, but a structured, persistent set of Markdown artifacts (CLAUDE.md, PRD.md, PLANS.md) residing in the local project directory. This local-first approach ensures that the state of the "machine" persists across sessions and crashes.¹¹

1.3 Progressive Disclosure as a Cognitive Firewall

A critical, often overlooked challenge in agentic systems is **Cognitive Overload**. This affects both the human user, who is inundated with streaming logs of agent "thoughts," and the AI agent, which degrades in performance when flooded with irrelevant context.

Atelier applies the design pattern of **Progressive Disclosure** to resolve this dual problem. For the human operator, the interface (constructed with TUI libraries like Bubble Tea) acts as a cognitive firewall. It reveals complexity only on demand—presenting a high-level status bar or a summary of the current phase initially, while reserving the ability to drill down into specific agent logs, diffs, or reasoning chains for debugging purposes.¹³

For the agent, Progressive Disclosure is implemented via **Context Curation**. An agent operating in the "Plan" phase is exposed only to the PRD and high-level architectural documents; it is shielded from the implementation details of unrelated modules. Tools are loaded dynamically based on the active "Skill" or phase: a "Database Agent" has access to SQL tools, while a "Frontend Agent" sees only React and CSS utilities. This strategy minimizes token usage, reduces the surface area for hallucination, and focuses the model's reasoning capabilities on the immediate task.²

1.4 The Economics of Autonomy

The shift to Atelier also introduces a new economic model for development. In a manual workflow, the cost of development is measured in human hours. In an agentic workflow, it is measured in **tokens** and **latency**. The architecture of Atelier is designed to optimize this economy. By breaking large, monolithic tasks into sequential, verifiable stages (the pipeline), we reduce the likelihood of catastrophic errors that require expensive "re-rolls" of the entire context window.

Furthermore, the system leverages a tiered model strategy. High-intelligence, high-cost models (like Claude 3.5 Opus) are reserved for the **Design** and **Review** phases, where reasoning depth is paramount. Faster, lower-cost models (like Claude 3.5 Sonnet or Haiku) are utilized for the **Gather** and **Build** phases, where speed and adherence to strict instructions are the primary requirements. This capabilities-matching ensures that the system is not only effective but also economically viable for continuous use.¹⁷

Part II: The Architecture of Atelier

2.1 The Control Plane: State Machines and Deterministic Hooks

The backbone of Atelier is a rigid **finite state machine (FSM)** that enforces the transition logic between the stages of the software development lifecycle (SDLC). Unlike a free-form chat interface where the user can erratic jump between topics, Atelier enforces a strict, linear progression: a feature cannot move to the **Design** phase until the artifacts of the **Specify** phase have been validated and locked.

This FSM is implemented using the **Claude Code Hooks** system.¹⁹ Hooks are deterministic shell scripts or Python functions that execute at specific lifecycle events (e.g., PreToolUse, PostToolUse, TaskCompleted). They act as the "physics" of the Atelier world, preventing agents from violating the laws of the workflow.

Table 1: Atelier State Machine Transitions and Guardrails

Current State	Trigger Action	Next State	Guardrail / Hook Mechanism
Idea	User Input (Natural Language)	Gather	None (Free brainstorming allowed).
Gather	UserApproval signal	Specify	PreCompact: Summarize context into CONTEXT.md to purge noise.
Specify	PRD_Finalized check	Design	PostToolUse: Run validate_prd.py schema check.
Design	TDD_Approved check	Plan	PostToolUse: Verify Architecture Decision Records (ADRs) exist.
Plan	Plan_Locked signal	Build	PreToolUse: Lock PRD (Read-Only), enable Edit/Bash tools.
Build	Tests_Pass signal	Review	TaskCompleted: Trigger automated test suite & linting.
Review	Code_Review_OK signal	Deploy	PostToolUse: Prevent direct push; trigger CI pipeline.

This architecture ensures **State Isolation**. When an agent is in the "Specify" phase, it operates within a dedicated session where write permissions are restricted strictly to PRD.md. It effectively *cannot* modify source code. This prevents the "rogue agent" problem, where an LLM attempts to "fix" a bug by changing the requirement rather than the implementation.²¹

2.2 The Data Plane: Local-First Artifacts

Atelier explicitly rejects the use of ephemeral vector databases or cloud-hosted session stores for primary state management. Instead, it relies on **Local-First File Storage**. Markdown files are chosen as the storage medium because they are human-readable, version-controllable (Git-native), and LLM-native (token-efficient).

The "Project State" is defined by a rigorous hierarchy of files residing in the `.atelier/` directory, which serves as the system's registry:

- **`.atelier/state.json`**: This file tracks the current phase of the FSM, active task IDs, agent assignments, and the pointers to the current active artifacts. It is the "save file" of the development session.
- **`.atelier/memory.md`**: A semantic log of high-level decisions, context updates, and user preferences. This acts as the long-term memory that persists across different claude sessions.
- **`.atelier/artifacts/`**: A versioned store for the immutable outputs of each phase (e.g., `reqs-v1.md`, `plan-v1.md`).
- **`.atelier/inboxes/`**: A set of JSON files used for asynchronous message passing between agents in the "Team" mode (discussed in Part IV).

This file-based approach enables robust **Session Resumption**. If the CLI crashes, the network fails, or the context window overflows, Atelier can "rehydrate" the agent's state simply by reading `state.json` and the relevant markdown artifacts. This ensures continuity without the need to re-process the entire conversation history, effectively solving the context window limitation for long-running tasks.¹¹

2.3 The Interface Layer: TUI Construction

The user interface of Atelier is constructed using **Bubble Tea** (for Go environments) or **Prompt Toolkit** (for Python environments), creating a rich, responsive Terminal User Interface (TUI). This layer wraps the raw claude CLI process, intercepting its input and output to render a structured workspace.

Key UI Components and Implementation:

- **The Canvas**: A split-screen view layout.
 - *Left Pane (Source of Truth)*: Displays the current driving artifact (e.g., the PRD during the Design phase, or the Plan during the Build phase). This is rendered using a markdown viewer component.
 - *Right Pane (Agent Stream)*: Displays the real-time execution logs of the agent.
- **The Input Bar**: A context-aware input field. In "Review" mode, it restricts input to strict "Approve/Reject/Comment" options to prevent scope creep. In "Idea" mode, it opens as a multi-line free-text field.
- **The Progress Indicator**: A visual representation of the Execution Plan (e.g., [x] Step 1, [] Step 2). This component parses the PLANS.md file in real-time to update the progress bar as the agent marks tasks as complete in the background.¹⁴

The TUI implements Progressive Disclosure by defaulting to a "Summary View." Instead of

streaming raw JSON tool calls or verbose internal monologue, the user sees high-level status updates (e.g., "Building Authentication Module... 45%"). The user can toggle a "Debug Mode" (e.g., via a keybinding like Ctrl+D) to expand the view, revealing the raw stdout and stderr streams from the underlying claude process for inspection.¹³

2.4 The Drivers: MCP and Tool Integration

Atelier uses the **Model Context Protocol (MCP)** to standardize how agents interact with the outside world. Rather than hard-coding integrations, Atelier loads MCP servers dynamically based on the CLAUDE.md configuration.

- **Filesystem Driver:** Allows safe, sandboxed access to read/write files.
- **Browser Driver:** Enables the "Gather" agent to scrape web documentation.
- **Git Driver:** Allows the "Deploy" agent to stage, commit, and push code.
- **Custom Drivers:** Atelier supports project-specific drivers, such as a PostgreSQL driver for database migrations or a Kubernetes driver for deployment verification.

This modular architecture allows Atelier to be agnostic to the specific tech stack of the project, adapting its capabilities by simply swapping out the MCP configuration files.¹⁰

Part III: The Pipeline – Phase by Phase Deep Dive

3.1 Phase 1: Idea \rightarrow Gather

Objective: Transform a vague, high-level user intent into a structured, comprehensive context package.

The workflow initiates in the **Idea** phase. The user provides a high-level goal, for example: "Add a subscription billing system using Stripe." Atelier immediately transitions to the **Gather** phase, activating the **Gather Agent**.

Mechanism:

This agent utilizes the **Gemini CLI** (for its large context window and search capabilities) or **Claude's Research Subagent**. It executes a series of search and read operations to compile the necessary context. Crucially, it does *not* possess write permissions for source code. Its sole output target is a CONTEXT.md file.

Strategic Insight:

A primary failure mode of autonomous agents is the lack of domain-specific knowledge. By enforcing a distinct "Gather" phase, Atelier ensures that the agent retrieves the most recent API documentation (e.g., Stripe API v2024-06-20) and analyzes the existing project's user model *before* it attempts to plan or design. This effectively performs "Retrieval Augmented Generation" (RAG) dynamically at the file system level, grounding the subsequent phases in reality.

Artifacts Produced:

- CONTEXT.md: A consolidated document containing scraped API docs, relevant existing code snippets, competitor analysis, and architectural constraints.

3.2 Phase 2: Specify (The Source of Truth)

Objective: Transmute the raw context into a rigorous, machine-parsable Product Requirements Document (PRD).

The **Specify Agent** consumes CONTEXT.md as its primary input. It engages in a **Socratic Dialogue** with the user to clarify ambiguities (e.g., "Do we need to handle pro-rated refunds?" or "What is the retry policy for failed webhooks?").

AI-Native PRD Schema:

Unlike traditional PRDs written for human consumption, the Atelier PRD is structured for machine parsing and logical consistency. It serves as the "Constitutional" document for the feature.

Table 2: AI-Native PRD Schema (Markdown Structure)

Section	Content Requirement	AI Function / Utility
1. Goal	Single sentence objective.	Anchors the system prompt for all subsequent agents.
2. Non-Goals	Explicit exclusions (e.g., "No PayPal support").	Prevents scope creep and "gold-plating."
3. User Stories	As a <role>, I want <action>, so that <benefit>.	Used to generate acceptance tests and verification steps.
4. Data Model	Schema definitions (SQL/JSON).	Guides the creation of database migration scripts.
5. Interface	API endpoints (OpenAPI spec) or UI component tree.	Defines contract tests and mock servers.
6. Verification	"Definition of Done" criteria (e.g., "Latency < 200ms").	Creates the assertion logic for the Review agent.

Validation Hook: At the end of this phase, Atelier triggers a deterministic validation script (validate_prd.py). This script parses the markdown to check for empty sections, ambiguous language (e.g., words like "fast," "easy," "user-friendly" are flagged for quantification), and logical contradictions. The pipeline is strictly blocked from advancing to the Design phase until this validation passes.²⁷

3.3 Phase 3: Design & Plan

Objective: Translate the PRD into a Technical Design Document (TDD) and a granular Execution Plan.

This phase splits the workflow into two parallel tracks, executed by specialized agents to ensure separation of concerns:

1. **The Architect Agent:** This agent creates the TDD (DESIGN.md). It makes high-level decisions on libraries, design patterns (e.g., "Use the Repository Pattern"), and error handling strategies. It explicitly maps the User Stories from the PRD to specific code modules and files.

2. **The Planner Agent:** This agent generates the **Execution Plan** (PLANS.md).

The "ExecPlan" Concept:

Derived from the "PLANS.md" methodology pioneered by Aaron Friel, the Execution Plan is a living document. It breaks the implementation work into atomic, verifiable steps. Crucially, **every step must be testable**.

ExecPlan Schema:

- **Phase 1: Scaffolding:** Create directories, config files. (Verification: Is output matches tree).
- **Phase 2: Core Logic:** Implement models and services. (Verification: Unit tests pass).
- **Phase 3: Integration:** Wire up API endpoints. (Verification: curl request returns 200 OK).
- **Decision Log:** A record of *why* certain technical choices were made (e.g., "Chose library X over Y because...").

Plan Mode vs. Build Mode: Atelier utilizes Claude Code's "Plan Mode" (read-only) for this phase. The agent is permitted to read files to understand the legacy system structure but is strictly blocked from writing code. This separation prevents the "premature optimization" trap where agents begin coding before the architecture is settled.²⁹

3.4 Phase 4: Build (The Swarm)

Objective: Execute the plan using a coordinated team of specialized agents.

This phase represents the engine room of Atelier, leveraging **Agent Teams** (specifically the TeammateTool enabled via CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS). The PLANS.md is fed into a **Task Dispatcher**, which assigns specific items to sub-agents.

Team Topology:

- **Lead Agent:** The orchestrator. It monitors PLANS.md, tracks overall progress, resolves dependencies, and manages the file locks.
- **Builder Agent:** The coder. It writes the implementation logic in src/.
- **Tester Agent:** The QA engineer. It writes tests in tests/ before the Builder acts, enforcing TDD.
- **Linter Agent:** The janitor. It runs static analysis, fixes formatting, and ensures style guide compliance.

Communication Architecture:

Agents do not communicate via shared memory buffers, which are prone to corruption.

Instead, they communicate via **Mailboxes**—local JSON files located in .atelier/teams/{name}/inboxes/.

- The Lead Agent writes a task directive: {"task_id": "1", "instruction": "Implement Stripe Wrapper", "context": "Refer to DESIGN.md section 2.1"}.
- The Builder Agent, running a loop that monitors its inbox, picks up the task, executes it, and replies with a status update and diff summary. This asynchronous, file-based messaging allows multiple agents to work in parallel without context collision.³⁰

The TDD Loop (SPARC Refinement):

Atelier enforces a strict **Red-Green-Refactor** loop:

1. **Red:** The Tester Agent writes a failing test case based on the TDD spec.
2. **Green:** The Builder Agent writes the minimum code necessary to pass that test.
3. **Refactor:** The Lead Agent reviews the code against the DESIGN.md constraints and requests optimizations or style fixes. This cycle is automated. If the tests fail more than 3 times, the Lead Agent halts the specific task and requests human intervention (an "Escalation Policy").⁶

3.5 Phase 5: Review & Deploy

Objective: Human-in-the-loop verification and safe deployment.

Review:

The **Review Agent** compiles a "Pull Request Summary" displayed in the TUI. It groups changes by feature, highlights high-risk areas (e.g., "Modified auth middleware," "Database schema change"), and presents the test coverage report. Progressive Disclosure is vital here: the user first sees the summary metrics; they can then expand sections to see specific file diffs.

Deploy:

Upon explicit user approval, Atelier triggers the deployment hook. This process is more than a simple git push. It involves:

- **Documentation Update:** The agent automatically updates README.md and API documentation to reflect the new features.
- **Cleanup:** It archives PLANS.md and CONTEXT.md into a history/ folder to maintain a clean workspace.
- **Commit:** It generates a semantic commit message based on the PRD ticket numbers (e.g., "feat: implement checkout flow (ref REQ-001)").
- **CI Trigger:** It pushes the branch and triggers the external CI/CD pipeline.³³

Part IV: The Control Plane – Technical Implementation

4.1 Orchestrating with CLAUDE.md and Custom Skills

The CLAUDE.md file serves as the "BIOS" or Constitution of the Atelier system. It resides in the project root and contains the immutable laws and behavioral directives that the agent must follow.

Snippet of an Atelier CLAUDE.md:

Atelier Directives

1. **Source of Truth:** You MUST NOT implement features not present in PRD.md.
2. **State Management:** Before starting any task, read .atelier/state.json. Update this file immediately upon task completion.
3. **Tool Safety:** You MUST run ./scripts/pre-check.sh before editing core config files.
4. **TDD Enforcement:** You MUST verify that a test file exists for a module before editing

the module implementation.

Custom Skills (SKILL.md):

Atelier defines specific "Skills" for each pipeline stage, stored in .atelier/skills/. These provide the agent with the procedural knowledge required for that phase.

- spec-architect: Instructs Claude on how to parse vague user intent into the rigid Atelier PRD schema.
- plan-generator: Teaches Claude to break down architectural requirements into atomic, idempotent execution steps (ExecPlans).
- test-engineer: Contains specific patterns for writing robust tests in the project's framework (e.g., Pytest, Jest), ensuring tests are not brittle.⁹

4.2 Deterministic Hooks for Governance

While LLMs are probabilistic engines, the software engineering workflow requires determinism. Hooks provide the bridge between these two worlds.

Key Hooks Configuration (.claude/settings.json):

- **PreToolUse (The Guard):**
 - *Trigger:* Any attempt to use Edit or Bash tools.
 - *Logic:* Check the current state in state.json. If the state is "Plan Mode" or "Specify," **BLOCK** the write operation. This mechanically enforces the read-only planning phase, preventing agents from "jumping the gun".³⁶
 - *Logic:* Check if the target file is in the "Protected List" (e.g., .env, production.config, main.tf). If so, block the action and require explicit human override.
- **PostToolUse (The Verifier):**
 - *Trigger:* Immediately after file edits.
 - *Logic:* Automatically execute the project's linter/formatter (e.g., prettier, black, gofmt). If the linter fails, the hook captures the error and feeds it back to the agent immediately as a "Tool Error." This forces the agent to self-correct its syntax without involving the human in a feedback loop.¹⁹
- **TaskCompleted (The Auditor):**
 - *Trigger:* When the agent signals that a task is done.
 - *Logic:* Scan the PLANS.md file. Ensure the corresponding checkbox [x] is marked. Run the specific test case associated with that task ID. If the test fails, **reject** the completion signal and force the agent to retry.²⁰

4.3 Agent Teams and Split-Pane Orchestration

Atelier leverages the TeammateTool (currently an experimental feature in Claude Code) to spawn and manage sub-processes.

Implementation Detail:

Atelier wraps the start command to initialize a tmux session, creating a physical separation of concerns in the terminal.

- **Pane 0 (Controller):** Runs the Atelier TUI and the **Lead Agent**. This is where the user

interacts.

- **Pane 1 (Worker A - Builder):** Runs an independent claude process restricted to the src/ directory.
- **Pane 2 (Worker B - Tester):** Runs an independent claude process restricted to the tests/ directory.

The Lead Agent orchestrates work via the shared file system. It writes a directive to tasks/builder/01.json. The Builder Agent, which is running a loop monitoring that folder, picks up the task, executes it, and writes the results to results/builder/01.json. The user sees a unified view in Pane 0, but the work happens in parallel. This isolates the context windows of the workers, preventing "Context Rot" (where the model forgets instructions due to token overload) and allowing for parallel execution of coding and testing.³⁰

Part V: The Data Plane – Artifact Schemas

5.1 The Product Requirements Document (PRD)

In the Atelier system, the PRD is not a passive PDF document; it is a **Constraint Object**.

Schema Structure:

Product Requirement: [Feature Name]

Metadata

- Status:
- Owner: [User Name]
- Agent Directives:

1. Context & Problem Statement

[Concise description of the user problem to be solved]

2. Functional Requirements (The Contract)

- REQ-001: System MUST [action] when [trigger].
- REQ-002: System MUST NOT [action].

3. Invariant Constraints

- Performance: Response time < 500ms p95.
- Security: No plain text storage of [field].

4. Verification Strategy

- Test Case 1: [Input] -> [Expected Output]

Second-Order Insight: By forcing requirements into numbered identifiers (REQ-001), Atelier allows the agent to link every line of code generated back to a specific requirement in the Git commit message (e.g., "feat: implement checkout flow (ref REQ-001)"). This creates full **Traceability** from the initial idea down to the specific lines of code.

5.2 The Execution Plan (PLANS.md)

This is the most critical artifact for autonomous execution. It bridges the gap between "what" needs to be built (PRD) and "how" it is built (Code).

Requirements for a Robust Plan:

1. **Idempotence:** Every step must be re-runnable without breaking the system.
2. **Verifiability:** Every step must define *how* the agent knows it is done (e.g., "grep output contains 'Success'").
3. **Dependency Management:** Explicitly stating which steps must precede others.

Schema:

Execution Plan:

Phase 1: Scaffolding

- [] 1.1 Create directory structure. (Verify: ls -R matches expected tree)
- [] 1.2 Initialize module config. (Verify: config file exists and parses)

Phase 2: Core Logic

- [] 2.1 Implement Domain Model. (Verify: Unit tests in tests/domain pass)
- [] 2.2 Implement Service Layer. (Verify: Mocked tests pass)

Decision Log

-: Chose library X over Y because. The "Decision Log" section is vital. It acts as a long-term memory for the project, preventing agents from re-litigating settled architectural decisions in future sessions.²⁹

Part VI: Implementation Guide & Workflows

6.1 Bootstrapping Atelier

To build Atelier, the user initiates a recursive setup where Claude Code is used to build the Atelier wrapper itself.

Step 1: The Constitution

Create the root CLAUDE.md. This defines the persona of Atelier itself: "You are Atelier, an orchestration engine. You do not write code unless explicitly in the Build Phase. You prefer planning over action."

Step 2: The Skill Library

Populate the .atelier/skills/ directory with core competencies. Commands like curl can be used to fetch best-practice templates for TDD and PRDs from a central repository.

Step 3: The TUI Wrapper

Write a simple Go wrapper (using Bubble Tea) that invokes the claude CLI.

- *Command:* atelier start [project_name]
- *Action:* Initializes the git repo, creates the .atelier directory structure, and launches the TUI.

6.2 The "Self-Healing" Workflow

One of Atelier's most powerful patterns is the **Self-Healing Verification Loop**.

1. **Detect:** The CI/CD pipeline (or a local pre-commit hook) detects a failure.
2. **Diagnose:** The Review Agent analyzes the stack trace and the recent changes to identify the root cause.
3. **Plan:** It updates PLANS.md with a specific "Remediation Phase."
4. **Fix:** The Builder Agent executes the fix.
5. **Verify:** The Tester Agent confirms the fix and ensures no regressions were introduced.

This loop runs autonomously. The user is only notified if the agent fails to fix the issue after \$N\$ attempts (the "Escalation Policy"), preventing infinite loops and token waste.³⁹

Part VII: Governance and Future Outlook

7.1 Managing Context Rot

A significant risk in long-running agentic workflows is **Context Rot**—the degradation of model performance as the context window fills with irrelevant logs and history. Atelier combats this via **Periodic Compaction**.

Compaction Strategy:

- At the end of every phase (e.g., after "Design"), Atelier triggers a summarize skill.
- The agent reads the entire verbose session log and distills it into a concise summary_v1.md.
- The session is effectively "rebooted" (a claude --restart equivalent is triggered).
- The new session loads *only* PRD.md, DESIGN.md, PLANS.md, and the new summary_v1.md. This strategy ensures that the "Signal-to-Noise Ratio" remains high throughout the project lifecycle, maintaining agent performance and reducing costs.⁴¹

7.2 The 2026 Horizon: Simul-Agentic Coding

Looking ahead to the near future, Atelier anticipates the trend of **Simul-Agentic Coding**. Rather than a strictly linear "Plan -> Build" flow, future versions will support asynchronous, real-time collaboration where human and agent work on the same files simultaneously.

- **Conflict Resolution:** Atelier will incorporate a "Merge Conflict Agent" capable of semantically understanding human edits vs. agent edits and merging them intelligently.

- **Predictive Pre-computation:** While the human is writing the specification, an ephemeral agent could be running in the background, testing feasibility hypotheses (e.g., "Can we actually query this API efficiently?") and flagging risks in real-time. This would effectively function as a "Spellchecker for Architecture".⁴³

Conclusion

Atelier represents the maturation of AI-assisted development. By wrapping the raw power of Claude code in a rigid, process-driven architecture, it transforms the stochastic nature of LLMs into a reliable engineering pipeline. It demands a shift in mindset: the developer is no longer the bricklayer, but the foreman. The artifact—the Specification, the Plan, the Test—becomes the primary product of human intellect, while the code becomes a derived commodity. In building Atelier, we are not just building a tool; we are defining the standard operating procedure for the software engineering workforce of the future.

Summary Table: The Atelier Pipeline Stack

Layer	Component	Technology	Responsibility
User	TUI	Bubble Tea (Go)	Progressive Disclosure, Approval Gates
Orchestration	FSM / Hooks	Bash / Python Scripts	Process Enforcement, State Transitions
Intelligence	Agents	Claude 3.5 Sonnet / Opus	Reasoning, Planning, Coding
Memory	Artifacts	Markdown / JSON	Persistence, Source of Truth
Execution	Kernel	Claude Code CLI	Tool Use (Bash, Edit, MCP)

End of Report

Works cited

1. What is agentic coding? How it works and use cases | Google Cloud, accessed on February 11, 2026, <https://cloud.google.com/discover/what-is-agentic-coding>
2. How Your Terminal Comes Alive with CLI Agents - InfoQ, accessed on February 11, 2026, <https://www.infoq.com/articles/agentic-terminal-cli-agents/>
3. Control Plane as a Tool: A Scalable Design Pattern for Agentic AI Systems - arXiv, accessed on February 11, 2026, <https://arxiv.org/html/2505.06817v1>
4. Agentic Engineering Is Here: How AI and CLI Tools Are Changing Software Development | by Dave Patten | Medium, accessed on February 11, 2026, <https://medium.com/@dave-patten/agentic-engineering-is-here-how-ai-and-cli-tools-are-changing-software-development-02f2c0727dbb>
5. The 2026 Guide to AI-Powered Test Automation Tools - DEV Community, accessed on February 11, 2026,

https://dev.to/matt_calder_e620d84cf0c14/the-2026-guide-to-ai-powered-test-automation-tools-5f24

6. SPARC Methodology · ruvnet/claude-flow Wiki - GitHub, accessed on February 11, 2026, <https://github.com/ruvnet/claude-flow/wiki/SPARC-Methodology>
7. ruvnet/sparc - GitHub, accessed on February 11, 2026, <https://github.com/ruvnet/sparc>
8. Claude Code overview - Claude Code Docs, accessed on February 11, 2026, <https://code.claude.com/docs/en/overview>
9. The Complete Claude Code CLI Guide - Live & Auto-Updated Every 2 Days - GitHub, accessed on February 11, 2026, <https://github.com/Cranot/claude-code-guide>
10. Code execution with MCP: building more efficient AI agents - Anthropic, accessed on February 11, 2026, <https://www.anthropic.com/engineering/code-execution-with-mcp>
11. Minimalist Claude Code Task Management Workflow | by Nick Tune ..., accessed on February 11, 2026, <https://medium.com/nick-tune-tech-strategy-blog/minimalist-claude-code-task-management-workflow-7b7bdcbc4cc1>
12. How to write a good spec for AI agents - Addy Osmani, accessed on February 11, 2026, <https://addyosmani.com/blog/good-spec/>
13. The Power of Progressive Disclosure in SaaS UX Design - DEV Community, accessed on February 11, 2026, <https://dev.to/lollypopdesign/the-power-of-progressive-disclosure-in-saas-ux-design-1ma4>
14. How to Build Command Line Tools with Bubbletea in Go - OneUptime, accessed on February 11, 2026, <https://oneuptime.com/blog/post/2026-01-30-how-to-build-command-line-tools-with-bubbletea-in-go/view>
15. Progressive Disclosure for Large Files UPDATED - Awesome Agentic Patterns, accessed on February 11, 2026, <https://agentic-patterns.com/patterns/progressive-disclosure-large-files/>
16. S01-MCP03: Progressive Disclosure for Knowledge Discovery in Agentic Workflows | by Prakash Kukanoor | Dec, 2025 | Medium, accessed on February 11, 2026, <https://medium.com/@prakashkop054/s01-mcp03-progressive-disclosure-for-knowledge-discovery-in-agentic-workflows-8fc0b2840d01>
17. Cloud Code 2.1.0: Multi-Agent AI Coding in the Terminal, accessed on February 11, 2026, <https://medium.com/@kapildevkhatik2/cloud-code-2-1-0-multi-agent-ai-coding-in-the-terminal-407f610c06ab>
18. Create custom subagents - Claude Code Docs, accessed on February 11, 2026, <https://code.claude.com/docs/en/sub-agents>
19. Claude Code Hooks: A Practical Guide to Workflow Automation - DataCamp, accessed on February 11, 2026, <https://www.datacamp.com/tutorial/claude-code-hooks>

20. Hooks reference - Claude Code Docs, accessed on February 11, 2026,
<https://code.claude.com/docs/en/hooks>
21. Deterministic AI Orchestration: A Platform Architecture for Autonomous Development, accessed on February 11, 2026,
<https://www.praetorian.com/blog/deterministic-ai-orchestration-a-platform-architecture-for-autonomous-development/>
22. Meridian — a zero-config way to give Claude Code a stable, persistent working environment inside your repo : r/ClaudeAI - Reddit, accessed on February 11, 2026,
https://www.reddit.com/r/ClaudeAI/comments/1own5qf/meridian_a_zeroconfig_way_to_give_claude_code_a/
23. Claude Code for Productivity Tasks, accessed on February 11, 2026,
<https://lucas-soares.medium.com/clause-code-for-productivity-tasks-ea51e993a890>
24. GopherCon 2023: Terminal UI Apps From the Ground Up with Bubble Tea - Andy Haskell, accessed on February 11, 2026,
<https://www.youtube.com/watch?v=B7QCo75GAwQ>
25. Progressive Disclosure design pattern, accessed on February 11, 2026,
<https://ui-patterns.com/patterns/ProgressiveDisclosure>
26. A practical guide to the architectures of agentic applications | Speakeasy, accessed on February 11, 2026,
[https://www.speakeeasy.com/mcp/using-mcp/ai-agents/architecture-patterns](https://www.speakeasy.com/mcp/using-mcp/ai-agents/architecture-patterns)
27. Write a PRD for a generative AI feature - Reforge, accessed on February 11, 2026,
<https://www.reforge.com/guides/write-a-prd-for-a-generative-ai-feature>
28. Product Requirements Document | AI Agent Tools - Beam AI, accessed on February 11, 2026, <https://beam.ai/skills/product-requirements-document>
29. Using PLANS.md for multi-hour problem solving, accessed on February 11, 2026,
https://developers.openai.com/cookbook/articles/codex_exec_plans/
30. Agent Teams: The Switch Got Flipped - Emergent Minds | paldo.dev, accessed on February 11, 2026, <https://paldo.dev/blog/agent-teams-the-switch-got-flipped/>
31. teammate-tool-implementation.md - GitHub Gist, accessed on February 11, 2026, <https://gist.github.com/sorrycc/4702f258f3d505495f4d5d984576a08d>
32. LLM4TDD: Best Practices for Test Driven Development Using Large Language Models - IEEE Xplore, accessed on February 11, 2026,
<https://ieeexplore.ieee.org/iel8/10734425/10734427/10734613.pdf>
33. Need for spec-kit self-assessment command to review and fix diverging non-conformant code #471 - GitHub, accessed on February 11, 2026,
<https://github.com/github/spec-kit/issues/471>
34. Building CI/CD for non-deterministic AI agents at scale - YouTube, accessed on February 11, 2026, <https://www.youtube.com/watch?v=Jfiq8qRuQAU>
35. Extend Claude with skills - Claude Code Docs, accessed on February 11, 2026,
<https://code.claude.com/docs/en/skills>
36. [Security Issue/Bug] Plan mode restrictions bypassed when spawning sub-agents #6527 - GitHub, accessed on February 11, 2026,
<https://github.com/anomalyco/opencode/issues/6527>

37. Feature Request: Add Hook for High-Level Task Completion (`TaskCompletion`) · Issue #4833 · anthropics/clause-code - GitHub, accessed on February 11, 2026,
<https://github.com/anthropics/clause-code/issues/4833>
38. Claude Code Agent Teams (Full Tutorial): The BEST FEATURE of Claude Code is HERE!, accessed on February 11, 2026,
<https://www.youtube.com/watch?v=zm-BBZIAJ0c>
39. Architect's Guide to Agentic Design Patterns: The Next 10 Patterns for Production AI, accessed on February 11, 2026,
<https://medium.com/data-science-collective/architects-guide-to-agnostic-design-patterns-the-next-10-patterns-for-production-ai-9ed0b0f5a5c3>
40. Agentic testing: Revolutionizing software QA automation - Blog - Synthesized.io, accessed on February 11, 2026, <https://www.synthesized.io/post/agentic-testing>
41. CLI reference - Claude Code Docs, accessed on February 11, 2026,
<https://code.claude.com/docs/en/cli-reference>
42. Designing multi-agent pipelines with shared state — how are you approaching it? - Models, accessed on February 11, 2026,
<https://discuss.huggingface.co/t/designing-multi-agent-pipelines-with-shared-state-how-are-you-approaching-it/171675>
43. Claude Code Multi-Agent Orchestration with Opus 4.6, Tmux and Agent Sandboxes, accessed on February 11, 2026,
https://www.youtube.com/watch?v=RpUTF_U4kiw
44. Getting Into Flow State with Agentic Coding - Kaushik Gopal's Website, accessed on February 11, 2026, <https://kau.sh/blog/agentic-coding-flow-state/>