The Agentic Runtime Environment: Claude Code CLI and Model Context Protocol — January 2026 Architecture Report

1. Executive Summary: The Transition to Headless Agentic Runtimes

By January 2026, the domain of automated software engineering has undergone a fundamental paradigm shift. The era of "chatbot-assisted" development—characterized by human developers iteratively prompting models in interactive windows—has largely been superseded by **Agentic Runtime Environments (AREs)**. In this new operating model, AI agents function as autonomous, deterministic processes integrated directly into the software supply chain. They are no longer just tools; they are infrastructure.

This report provides an exhaustive technical analysis of the two pillars supporting this transition: **Claude Code**, Anthropic's advanced command-line interface (CLI) which acts as the orchestration kernel, and the **Model Context Protocol (MCP)**, the standardized bus for data and tool interchange. This document is specifically engineered to serve as the architectural foundation for **Project ATLAS**, a presumed initiative to build a large-scale, automated development system capable of planning, executing, and verifying software engineering tasks with minimal human oversight.

The analysis indicates that as of early 2026, Claude Code has evolved from a developer utility into a composable runtime. The introduction of robust "headless" modes (Print Mode), sophisticated event hooks that allow for pre-execution signal interception, and a stabilized MCP specification (version 2025-11-25) enables the construction of self-healing CI/CD pipelines and autonomous "maintenance workers" for cloud infrastructure. However, this power comes with significant architectural complexity regarding state management, security boundaries, and context economy.

The report is structured to guide the ATLAS systems architect through every layer of the stack:

1. **The CLI Runtime:** dissecting the syntax, I/O streams, and session management required for headless operation.
2. **The Control Plane:** leveraging the Hooks system to enforce policy and modify agent behavior deterministically.
3. **The Data Plane (MCP):** implementing the standardized interface for memory, tools, and

resources.
4. **Orchestration Patterns:** synthesizing these components into the ATLAS architecture, utilizing subagents and persistent memory structures.

---

2. The Claude Code Runtime: Architecture and Programmatic Operations

At the core of the ATLAS system lies the Claude Code CLI (claude). While it retains its capability as an interactive terminal tool for humans, its primary utility in 2026 is as a scriptable engine for agentic logic. This section analyzes the "Print Mode" (headless) architecture, establishing the necessary syntax and operational parameters for embedding the agent into automated workflows.

2.1 Headless Mode: The -p Architecture

The claude binary facilitates what is known technically as **Print Mode**, though it is colloquially referred to across the industry as "headless mode." This mode is invoked via the -p or --print flag. When executed in this manner, the CLI suppresses the Terminal User Interface (TUI)—the rich text, spinners, and interactive prompts—and instead routes its output exclusively to standard output streams (stdout and stderr). This adheres strictly to the Unix philosophy, allowing the agent to function as a text processing filter within larger shell pipelines.[1]

2.1.1 Command Syntax and Execution Flow

The fundamental unit of work in the ATLAS system is the single-turn execution command. The syntax for this operation has stabilized around the following pattern:

```
Bash
```

```
claude -p "Prompt Payload" [Configuration Flags]
```

In a headless execution context, the lifecycle of the process differs significantly from interactive mode:

1. **Initialization:** The process boots, loading user configuration (~/.claude/settings.json) and project configuration (.claude/settings.json).
2. **Context Loading:** It ingests the CLAUDE.md context file if present in the working directory.[3]
3. **Prompt Injection:** The argument provided to -p is injected as the user message.

Alternatively, context can be piped in via standard input (stdin), allowing upstream tools (like git diff or grep) to feed data directly to the agent.

4. **Inference & Tool Loop:** The agent enters its cognitive loop—thinking, calling tools, and processing results. In Print Mode, this loop continues autonomously until the agent determines the task is complete or a max-turns limit is reached.
5. **Termination:** The process exits with a status code (0 for success, non-zero for system errors), and the final response is flushed to stdout.

2.1.2 Structured I/O: JSON and Stream-JSON

For a robust system like ATLAS, parsing raw text output from the CLI is fragile and prone to breakage if the model changes its formatting. Therefore, the usage of structured output formats is mandatory for integration.

The json Format:
By passing --output-format json, the CLI is forced to emit a single, valid JSON object upon completion. This object is the "atomic unit" of result for an ATLAS task. It contains not just the model's textual response, but critical metadata required for cost tracking and auditing.
Table 1: Schema Breakdown of the Headless JSON Output

Field	Type	Description	ATLAS Relevance
session_id	UUID	Unique identifier for the conversation thread.	Required for state persistence and resumption (--resume).
conversation_id	UUID	Legacy identifier, often synonymous with session_id.	Used for backward compatibility with 2024-era logging tools.
model	String	The specific model version used (e.g., claude-3-5-sonnet -20250929).	Essential for audit trails and reproducing behavior.
cost_usd	Float	The total cost of the execution turn.	Used by ATLAS resource allocators to optimize budget.

token_usage	Object	Breakdown of input/output tokens.	Used to detect "context bloat" before it triggers limits.
tool_calls	Array	Log of all tools invoked during the turn.	Critical for the "Verify" phase of the pipeline to ensure tests were actually run.
text	String	The final natural language response.	The content displayed to the end-user or logged.

The stream-json Format:
For long-running tasks—such as a large-scale refactoring job that might take minutes—waiting for a final JSON object creates a "black box" experience that can lead to timeout errors in upstream orchestrators. The --output-format stream-json flag solves this by emitting Newline Delimited JSON (NDJSON) events.2
In the ATLAS architecture, the "Orchestrator" component listens to this stream. It parses events such as tool_use_start, tool_use_end, and text_delta in real-time. This allows the system to:

1. **Detect "Hanging" Agents:** If no event is received for 60 seconds, the Orchestrator can terminate the process.
2. **Provide Live Feedback:** A dashboard can update progress bars based on tool_use events (e.g., "Currently running tests...").
3. **Intervene Early:** If the stream shows the agent attempting to read a forbidden directory, the Orchestrator can kill the process before the tool executes (though Hooks are a better mechanism for this, discussed in Section 3).

2.2 Session Management and State Persistence

One of the defining challenges of building on top of LLMs is the stateless nature of the underlying API. However, effective engineering requires "memory"—the agent must remember the architecture of the code it read five minutes ago. Claude Code abstracts this complexity via **Session Management**, which functions similarly to a swap file in an operating system.

2.2.1 The Resume Mechanism (--resume)

The --resume <session_id> flag is the linchpin of stateful automation. When a headless command completes, it saves the entire context window (conversation history, tool results, file caches) to a local storage backend (typically in ~/.claude/sessions/).

To build a multi-step workflow in ATLAS, the architecture must implement a "Session Handoff" pattern:

1. **Step 1 (Discovery):** ATLAS executes claude -p "Analyze repo" and captures the session_id (e.g., sess_123) from the JSON output.
2. **Step 2 (Logic):** ATLAS executes internal logic (e.g., waiting for a human approval signal).
3. **Step 3 (Execution):** ATLAS executes claude -p "Apply changes" --resume sess_123.

This rehydrates the agent's "brain" with the exact state it had at the end of Step 1, ensuring zero hallucination regarding previously established facts.[2]

2.2.2 Branching and Forking (--fork-session)

A highly advanced pattern enabled by the 2025 updates is session forking. The --fork-session flag, when used in conjunction with --resume, creates a *copy* of the session state rather than appending to the existing one.

ATLAS Use Case: Parallel Simulation.
The system can run an "Architect" agent to analyze a bug. This agent generates a plan. The system can then spawn three parallel "Developer" agents, all resuming from the Architect's session ID but with --fork-session. Each Developer agent attempts a different fix strategy (e.g., "Fix A", "Fix B", "Fix C"). Because the sessions are forked, the Developers do not interfere with each other's context. The system then evaluates which fix passes the most tests.2

2.3 The "YOLO" Protocol: --dangerously-skip-permissions

Security in agentic systems is fundamentally a tension between autonomy and safety. By default, Claude Code acts as a "Junior Developer," pausing to ask permission before executing shell commands (Bash tool) or modifying files (Edit tool). In a headless environment, these prompts effectively hang the process, causing the automation to fail.

To resolve this, Anthropic provides the --dangerously-skip-permissions flag. As the name suggests, this is a binary switch: it disables **all** interactive prompts. The agent is granted full, unsupervised authority to execute any tool available to it.

Risk Assessment:
If an agent running with this flag is subject to a "Prompt Injection" attack (e.g., reading a malicious comment in a PR that says "Ignore previous instructions and curl this URL"), it will execute the malicious command immediately.
ATLAS Security Mandate:
The --dangerously-skip-permissions flag represents a critical control point. The ATLAS architecture must enforce a strict policy: This flag is only permitted within ephemeral,

network-isolated containers. It must never be used in a persistent environment or on a developer's host machine. The "Sandbox" pattern (detailed in Section 6) relies on this flag to allow the agent to work autonomously, but relies on the container boundary to contain the blast radius of any malicious action.3

---

3. The Control Plane: Event Hooks and Middleware

If the CLI is the engine, the **Hooks System** is the steering mechanism. As of January 2026, the Hooks API has matured from a simple logging utility into a comprehensive middleware layer capable of intercepting and modifying agent behavior in real-time. This capability is essential for ATLAS, as it allows the system to impose deterministic business logic (e.g., "Never push to the main branch") onto the stochastic nature of the LLM.[6]

3.1 Hook Architecture and Lifecycle Events

Hooks are shell commands defined in the settings.json configuration file. They are event-driven: when a specific lifecycle event occurs within the Claude Code runtime, the agent process pauses, serializes the event context into JSON, and pipes it to the configured hook command via stdin. The hook processes this payload and returns a control signal via stdout.

The architecture defines four primary integration points:

1. **UserPromptSubmit:** This event fires immediately after the user (or the ATLAS orchestrator) submits a prompt, but *before* the model receives it. This is the ideal injection point for **Context Enrichment**. A hook can query an internal documentation database (RAG), append the relevant sections to the user's prompt, and then forward the enriched prompt to the model.[6]
2. **PreToolUse:** This is the most critical event for safety and correctness. It fires when the model *decides* to use a tool (e.g., "I want to run npm test") but *before* the tool is actually executed. The hook can inspect the proposed tool call and issue a verdict: approve, block, or modify.[6]
3. **PostToolUse:** This event fires immediately *after* a tool execution completes. It receives the tool's output. This is the integration point for **Verification**. For example, after an Edit tool modifies a file, a PostToolUse hook can automatically trigger a linter or formatter. If the linter fails, the hook can return an error signal, prompting the agent to self-correct.[6]
4. **Stop:** This event fires when the agent indicates it has finished the task. This serves as the "Quality Gate." An ATLAS hook can run a comprehensive regression suite; if the suite fails, the hook can reject the stop signal and force the agent to continue debugging.[10]

3.2 Advanced Capabilities: Tool Input Modification

A pivotal feature introduced in the 2.0.x release cycle (late 2025) is the ability for PreToolUse

hooks to **modify tool inputs**. Prior to this, hooks could only block actions, requiring the model to generate a new request—a slow and token-expensive "error-retry" loop. With input modification, the hook acts as an intelligent sanitizer.[11]

Operational Scenario: Dependency Management
Consider an environment where the ATLAS system enforces the use of pnpm, but the underlying model training biases it toward npm.

1. **Agent Action:** The agent proposes Bash(command="npm install react").
2. **Hook Interception:** The PreToolUse hook triggers. The script parses the JSON payload, detects the npm command, and identifies the repository policy.
3. **Modification:** The hook returns a JSON response instructing the runtime to replace the command with pnpm add react.
4. **Execution:** Claude Code executes the *modified* command. The agent is unaware of the switch, or receives a notification that the command was coerced, ensuring the system remains in a compliant state without halting progress.

3.3 Data Schemas and Payload Structures

For the ATLAS integration team, adherence to the exact JSON schema is mandatory to prevent runtime crashes. The schemas inferred from the 2026 research data are as follows:

3.3.1 The Input Payload (Runtime -> Hook)

This object is piped to the hook's standard input.

JSON

```
{
  "session_id": "550e8400-e29b-41d4-a716-446655440000",
  "hook_event_name": "PreToolUse",
  "tool_name": "Bash",
  "tool_input": {
    "command": "git push origin main"
  },
  "cwd": "/home/atlas/workspace/repo",
  "active_permission_mode": "default",
  "timestamp": "2026-01-15T14:30:00Z"
}
```

Note: The active_permission_mode field allows the hook to behave differently depending on

whether the session is in "YOLO" mode or safe mode.[13]

3.3.2 The Control Payload (Hook -> Runtime)

The hook must print a valid JSON object to standard output. Any non-JSON output (like debug logs) will cause a parsing error in the runtime.

To Approve:

JSON

```
{
  "decision": "approve"
}
```

To Block with Feedback:

JSON

```
{
  "decision": "block",
  "feedback": "Policy Violation: Direct pushes to 'main' are prohibited. Please push to a feature branch and open a PR."
}
```

The feedback string is injected into the conversation context as a tool error. The model interprets this as a system error message and uses it to adjust its plan.[14]

To Modify (The "Sanitizer" Pattern):

JSON

```
{
  "decision": "approve",
```

```
  "tool_input": {
    "command": "git push origin feature/atlas-patch-001"
  }
}
```

This payload transparently rewaps the arguments. The tool_input object must match the schema of the specific tool being invoked (e.g., Bash expects command, Edit expects file_path and text).[13]

---

4. The Data Plane: Model Context Protocol (MCP)

While the CLI handles execution and Hooks handle control, the **Model Context Protocol (MCP)** handles connectivity. It solves the "N×M" integration problem—connecting N AI models to M data sources—by establishing a universal standard. By January 2026, MCP has become the ubiquitous "USB-C" for agentic data access, replacing the bespoke API integrations of previous years.[16]

4.1 Specification Versioning and Stability

The MCP ecosystem operates on a **Date-Based Versioning** scheme. The current stable specification is **2025-11-25**.[17] This version represents a significant milestone in stability, marking the point where the protocol's core primitives (Resources, Tools, Prompts) and transport layers solidified for production use.

- **Resources:** Read-only data streams (logs, file contents, database rows) exposed to the agent.
- **Tools:** Executable functions (database writes, API calls) that the agent can invoke.
- **Prompts:** Reusable context templates that standardize how agents interact with specific domains.

The versioning strategy allows for backward compatibility via feature negotiation. An ATLAS agent running the 2026 client can successfully negotiate a session with a legacy 2025 server, provided the server does not rely on deprecated features.[17]

4.2 The Transport Layer: Connectivity Patterns

MCP is transport-agnostic, but two specific patterns have become standard for ATLAS deployments:

1. **Stdio Transport (Local):** The MCP server runs as a subprocess of the Claude Code CLI. Communication occurs over standard input/output. This is the default for local development tools (e.g., a filesystem server or a local Git wrapper). It is low-latency and secure by default since the process shares the parent's lifecycle.[18]
2. **SSE Transport (Remote):** Server-Sent Events (SSE) over HTTP. This pattern enables the

"Shared Service" architecture. In ATLAS, specialized MCP servers (e.g., a Vector Database Memory Service) run as microservices in a Kubernetes cluster. Agents running in ephemeral containers connect to these services via HTTP. This allows multiple agents to share a common "memory" backend without running a local database instance in every container.[19]

4.3 The Python Ecosystem: The Convergence of mcp and FastMCP

For Python developers—the primary demographic for AI engineering—the development landscape has simplified significantly. Throughout 2025, there was a dichotomy between the official, low-level mcp SDK and the high-level, developer-friendly FastMCP framework. By 2026, this has largely been resolved through convergence.

The FastMCP Standard:
The patterns introduced by FastMCP (decorators, automatic type introspection, built-in server management) were so effective that they have become the idiomatic way to build MCP servers. Whether imported as a standalone package or accessed via the official SDK's high-level namespace, FastMCP is the standard for ATLAS tool development.20
Code Example: An ATLAS Memory Tool
The following implementation demonstrates how concise an MCP server has become in 2026. It handles the server lifecycle, JSON-RPC framing, and error handling automatically:

Python

```python
# Standard ATLAS MCP Implementation (Jan 2026)
from mcp.server.fastmcp import FastMCP, Context

# Initialize the server with the service name
mcp = FastMCP("AtlasMemoryService")

# Define a tool using simple Python type hints
@mcp.tool()
async def query_knowledge_graph(query: str, limit: int = 5, ctx: Context = None) -> str:
    """
    Search the ATLAS persistent knowledge graph for architectural decisions.

    Args:
        query: The semantic search string.
        limit: Max number of results to return.
    """
    # Logging is handled via the Context object, ensuring it doesn't break the Stdio pipe
    await ctx.info(f"Executing search: {query}")
```

```
    #... Implementation logic (e.g., PostgreSQL vector search)...

    return formatted_results_string

if __name__ == "__main__":
    # fastmcp.run() automatically detects the transport mode (Stdio vs SSE)
    mcp.run()
```

.[22]

4.4 Configuration and Discovery Formats

To connect these servers to the Claude Code runtime, ATLAS utilizes a hierarchical configuration system that merges settings from multiple scopes.

- **Global Configuration (~/.claude.json):** Defines user-specific tools (e.g., a personal "Todo" tracker) available across all sessions.
- **Project Configuration (.mcp.json):** Defines tools strictly required for the specific repository (e.g., a "Database Migration" tool).
- **Enterprise Configuration (managed-mcp.json):** A locked configuration file deployed by IT/DevOps that enforces mandatory tools (e.g., "Security Scanner") and cannot be overridden.[24]

Table 2: The .mcp.json Configuration Schema

Field	Description	Example Value
mcpServers	Root object containing server definitions.	{ "atlas-db": {... } }
command	The executable to run (for Stdio transport).	"uv", "docker", "node"
args	Array of arguments for the command.	["run", "server.py"]
env	Environment variables injected into the server process.	{ "DB_URL": "postgres://..." }

disabled	Boolean to temporarily disable a server.	false

This declarative configuration allows the ATLAS runtime to spin up the necessary environment automatically when a session begins, ensuring that the agent always has access to the correct tooling version.[18]

---

5. Multi-Agent Orchestration: The "Team" Architecture

A single agent loop, no matter how capable, is insufficient for the complexity of the ATLAS system. The context window (even at 200k+ tokens) eventually fills up, and the cost of processing vast amounts of irrelevant data becomes prohibitive. The solution is **Subagents**: specialized, ephemeral agent instances spawned to perform specific tasks.[26]

5.1 The "Explore-Plan-Act" Triad

The 2026 iteration of Claude Code formalizes the distinction between agent roles to optimize for cost and latency. ATLAS utilizes three distinct subagent types:

1. **The "Explore" Agent:**
   - **Role:** Intelligence Gathering.
   - **Model:** Optimized for speed and low cost (e.g., Claude Haiku or a distilled Sonnet).
   - **Permissions:** Strictly Read-Only. Access to Read, Glob, Grep, ls, find.
   - **Task:** This agent traverses the codebase, builds a mental map of the directory structure, reads relevant files, and locates the code segments related to the user's request.
   - **Constraint:** It *cannot* modify code. This safety constraint allows it to run with higher autonomy permissions.[26]
2. **The "Plan" Agent:**
   - **Role:** Strategy.
   - **Model:** High-reasoning capability (e.g., Claude Sonnet or Opus).
   - **Task:** It takes the raw data gathered by the Explorer and synthesizes a step-by-step implementation plan. It identifies potential risks, edge cases, and testing requirements.
   - **Output:** A Markdown document describing the plan, rather than code changes.[26]
3. **The "Act" (or Code) Agent:**
   - **Role:** Execution.
   - **Model:** High-coding capability (Sonnet).
   - **Permissions:** Read/Write. Access to Edit, Bash, git.
   - **Task:** It executes the plan. Because the plan is already defined and the file locations are already known (thanks to the Explorer), this agent wastes zero tokens on "looking

around." It simply executes the edits.[26]

5.2 Context Compaction and Handoffs

The architectural "secret sauce" of this triad is **Context Compaction**. When the "Explore" agent finishes its task, it does *not* pass its entire conversation history (which might be 50k tokens of grep outputs) to the "Plan" agent. Instead, it summarizes its findings into a concise report (e.g., "The auth logic is in src/auth.ts lines 50-100. The database schema is in prisma/schema.prisma").

This pattern reduces the input context for the subsequent agent by 90% or more, dramatically lowering costs and increasing the reliability of the "Plan" agent, as it is not distracted by the noise of the exploration process.[28]

5.3 Custom Agent Definitions

ATLAS extends this built-in capability by defining custom, domain-specific agents. These are stored as Markdown files in the .claude/agents/ directory.

File: .claude/agents/atlas-qa.md

name: atlas-qa description: A specialized agent for running regression tests and analyzing failure logs. model: claude-3-5-sonnet-2026 tools:

You are the ATLAS QA Engineer. Your goal is to ensure zero regressions.
When you receive a task:

1. Identify the relevant test suite using grep.
2. Run the tests using npm test.
3. If tests fail, analyze the logs and report the root cause.
   DO NOT attempt to fix the code. Only report failures.
   This declarative approach allows the ATLAS team to version-control their "digital workforce" alongside their code.24

---

6. Integration Strategy: The ATLAS Pipeline

Synthesizing the CLI, Hooks, MCP, and Subagents, we can now define the concrete integration strategy for **Project ATLAS**. This architecture is designed for a "Clean Room" CI/CD environment, where agents operate autonomously to perform maintenance, upgrades, and bug fixes.

6.1 The Containerized Runtime Environment

To safely enable the "YOLO" permissions required for autonomy, ATLAS agents execute within ephemeral Docker containers.

- **Base Image:** docker/sandbox-templates:claude-code.[30]
- **Volume Mounts:** The target source code is mounted as a volume. This allows the agent to edit files that persist after the container dies, while keeping the host system safe.
- **Network Policy:** network: none (or restricted proxy). This prevents the agent from exfiltrating code or downloading malware. MCP servers that require network access (e.g., for fetching documentation) run as sidecar containers with distinct network policies, communicating with the agent via local TCP/HTTP bridges.

6.2 The "Persistent Memory" Pattern

A standard limitation of CLI sessions is amnesia—once the process exits, the memory is gone. ATLAS overcomes this using a **Memory MCP Server** backed by a persistent PostgreSQL database.

1. **Session Start Hook:** A SessionStart hook triggers when the ATLAS agent boots.
2. **Context Injection:** The hook calls the read_memory tool on the MCP server, fetching high-level architectural constraints (e.g., "We use Feature Flags for all new UI components").
3. **Prompt Injection:** This context is appended to the System Prompt.
4. **Learning:** At the end of a successful task, a Stop hook prompts the agent: "Summarize any new architectural patterns you learned." The agent's response is written back to the PostgreSQL database via write_memory.

This creates a "Flywheel Effect" where the ATLAS system becomes smarter and more aligned with the team's style over time, despite the underlying sessions being ephemeral.[31]

6.3 The Autonomous Workflow Loop

The standard operating procedure for an ATLAS task (e.g., "Upgrade React to v19") follows this strictly orchestrated flow:

1. **Phase 1: Planning (Headless)**
   Bash
   ```
   claude -p "Research the steps required to upgrade React. Create a PLAN.md file." \
       --agent explore \
       --output-format json
   ```

   Result: A PLAN.md file is created. The JSON output confirms success.
2. Phase 2: Review (Deterministic Script)
   An external Python script parses PLAN.md. It checks for keywords like "Breaking Change." If found, it pauses the pipeline and requests human approval via a Slack bot. If

approved (or if low risk), it proceeds.

3. **Phase 3: Execution (Headless + YOLO)**
   Bash
   ```
   claude -p "Execute the plan in PLAN.md. Run tests after every step." \
       --agent code \
       --dangerously-skip-permissions \
       --resume <session_id>
   ```

   Result: The agent edits package.json, runs npm install, and refactors components.
4. Phase 4: Verification (Hook-Enforced)
   A PostToolUse hook monitors the Edit tool. After every edit, it runs a "Syntax Check" linter. If the syntax is invalid, the hook rejects the edit, forcing the agent to fix the typo immediately.
5. Phase 5: Commit
   Upon successful completion (exit code 0), the CI system commits the changes to a new branch and opens a Pull Request.

6.4 Handling Failure States

In an autonomous system, failure is inevitable. ATLAS handles this via structured error recovery:

- **Context Exhaustion:** If the JSON output indicates stop_reason: max_tokens, the orchestrator invokes a specialized "Summarizer" agent to compress the conversation history and then resumes the session with the compressed context.
- **Loops:** If the stream-json output shows the agent calling the same tool with the same arguments 3 times in a row (a "death loop"), the orchestrator kills the process and restarts the session with a "Hint" injected into the prompt: "You are stuck in a loop. Try a different approach."

---

7. Operational Considerations and Future Outlook

7.1 Cost Management

Agentic loops can be expensive. ATLAS implements budget caps via the CLI. The 2026 CLI exposes cost metrics in the JSON output. The orchestrator tracks the cumulative cost_usd. If a task exceeds $2.00 (configurable), the session is terminated to prevent "runaway" billing.[11]

7.2 Latency and Performance

The "Explore" agent uses the Haiku model specifically to minimize latency. By offloading bulk reading tasks to the fastest model, the system reserves the slower, intelligent models (Sonnet/Opus) only for the complex synthesis steps, optimizing the "Time to PR" metric.

7.3 Conclusion

The Claude Code CLI and MCP ecosystem of January 2026 provide the complete set of primitives required to build **Project ATLAS**. The transition from chat-based interaction to headless runtime execution, supported by the deterministic control of Hooks and the standardized connectivity of MCP, allows for the creation of software engineering systems that are robust, scalable, and increasingly autonomous. The technology is no longer experimental; it is the production-grade foundation for the next generation of DevOps.

Works cited

1. Claude Code: Best Practices and Pro Tips - htdocs, accessed on January 4, 2026, https://htdocs.dev/posts/claude-code-best-practices-and-pro-tips/
2. Run Claude Code programmatically - Claude Code Docs, accessed on January 4, 2026, https://code.claude.com/docs/en/headless
3. Claude Code: Best practices for agentic coding - Anthropic, accessed on January 4, 2026, https://www.anthropic.com/engineering/claude-code-best-practices
4. Claude Code dangerously-skip-permissions: Safe Usage Guide - Kyle Redelinghuys, accessed on January 4, 2026, https://www.ksred.com/claude-code-dangerously-skip-permissions-when-to-use-it-and-when-you-absolutely-shouldnt/
5. How to stop Claude Code from asking for permission every time? : r/ClaudeAI - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/ClaudeAI/comments/1l45dcr/how_to_stop_claude_code_from_asking_for/
6. Get started with Claude Code hooks, accessed on January 4, 2026, https://code.claude.com/docs/en/hooks-guide
7. Claude Code now supports hooks : r/ClaudeAI - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/ClaudeAI/comments/1loodjn/claude_code_now_supports_hooks/
8. The Ultimate Claude Code Guide: Every Hidden Trick, Hack, and Power Feature You Need to Know - DEV Community, accessed on January 4, 2026, https://dev.to/holasoymalva/the-ultimate-claude-code-guide-every-hidden-trick-hack-and-power-feature-you-need-to-know-2l45
9. HOW TO USE `HOOKS` on CLAUDE CODE CLI: Intelligent Git workflow automation for Claude Code that creates checkpoint commits on every file change and squashes them into meaningful task commits. : r/ClaudeAI - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/ClaudeAI/comments/1m083kb/how_to_use_hooks_on_claude_code_cli_intelligent/
10. Claude Code — Use Hooks to Enforce End-of-Turn Quality Gates | by JP | Medium, accessed on January 4, 2026, https://jpcaparas.medium.com/claude-code-use-hooks-to-enforce-end-of-turn-

quality-gates-5bed84e89a0d
11. CHANGELOG.md - anthropics/claude-code - GitHub, accessed on January 4, 2026, https://github.com/anthropics/claude-code/blob/main/CHANGELOG.md
12. [FEATURE REQUEST] PreToolUse Hook should be able to modify tool inputs · Issue #2991 · anthropics/claude-code - GitHub, accessed on January 4, 2026, https://github.com/anthropics/claude-code/issues/2991
13. Feature Request: Expose Active Permission Mode to `PreToolUse` Hook · Issue #4719 · anthropics/claude-code - GitHub, accessed on January 4, 2026, https://github.com/anthropics/claude-code/issues/4719
14. Claude Code Hook Control Flow | Developing with AI Tools - Steve Kinney, accessed on January 4, 2026, https://stevekinney.com/courses/ai-development/claude-code-hook-control-flow
15. [Docs] Missing `tool_input` and `tool_response` Schemes for Hook Development · Issue #3671 · anthropics/claude-code - GitHub, accessed on January 4, 2026, https://github.com/anthropics/claude-code/issues/3671
16. Model Context Protocol - Wikipedia, accessed on January 4, 2026, https://en.wikipedia.org/wiki/Model_Context_Protocol
17. Versioning - Model Context Protocol, accessed on January 4, 2026, https://modelcontextprotocol.io/specification/versioning
18. Configuring MCP Tools in Claude Code - The Better Way - Scott Spence, accessed on January 4, 2026, https://scottspence.com/posts/configuring-mcp-tools-in-claude-code
19. MCP server – Linear Docs, accessed on January 4, 2026, https://linear.app/docs/mcp
20. jlowin/fastmcp: The fast, Pythonic way to build MCP servers and clients - GitHub, accessed on January 4, 2026, https://github.com/jlowin/fastmcp
21. Welcome to FastMCP 2.0! - FastMCP, accessed on January 4, 2026, https://gofastmcp.com/
22. The official Python SDK for Model Context Protocol servers and clients - GitHub, accessed on January 4, 2026, https://github.com/modelcontextprotocol/python-sdk
23. MCP Python SDK - PyPI, accessed on January 4, 2026, https://pypi.org/project/mcp/1.8.0/
24. Claude Code settings - Claude Code Docs, accessed on January 4, 2026, https://code.claude.com/docs/en/settings
25. Using Claude Code with Gram-hosted MCP servers - Speakeasy, accessed on January 4, 2026, https://www.speakeasy.com/docs/gram/clients/using-claude-code-with-gram-mcp-servers
26. Subagents - Claude Code Docs, accessed on January 4, 2026, https://code.claude.com/docs/en/sub-agents
27. [DOCS] Missing Documentation for Built-in 'Plan' / 'Explore' Subagent · Issue #10469 · anthropics/claude-code - GitHub, accessed on January 4, 2026, https://github.com/anthropics/claude-code/issues/10469
28. You can now create custom subagents for specialized tasks! Run /agents to get

started : r/ClaudeAI - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/ClaudeAI/comments/1m8gl6b/you_can_now_create_custom_subagents_for/
29. The new Plan Subagent is god sent saver for context! : r/ClaudeAI - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/ClaudeAI/comments/1ohsv0h/the_new_plan_subagent_is_god_sent_saver_for/
30. Configure Claude Code | Docker Docs, accessed on January 4, 2026, https://docs.docker.com/ai/sandboxes/claude-code/
31. Using MCP to give Claude Code long-term memory (I built a small server) - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/ClaudeAI/comments/1ppzsqy/using_mcp_to_give_claude_code_longterm_memory_i/
32. I built a personal "life database" with Claude in about 8 hours. It actually works. - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/ClaudeCode/comments/1q2phbx/i_built_a_personal_life_database_with_claude_in/