Architectural Blueprint for a Unified Agentic Sandbox Integration across Modern Coding Interfaces

The contemporary landscape of software engineering is undergoing a transformative shift from human-centric integrated development environments to agent-centric runtimes. In this new paradigm, the terminal serves not merely as a shell for command execution, but as the primary coordination layer for a fleet of autonomous AI agents. The central challenge in this evolution is the provisioning of secure, isolated, and scalable environments where these agents can operate with the full agency of a human engineer without endangering the host system's integrity. The implementation of agent sandboxes—ephemeral, programmable compute units—represents the definitive solution to this challenge. This report delineates the architectural requirements for a unified CLI starter kit that integrates agent sandboxes across three dominant platforms: Claude Code, Gemini CLI, and GitHub Copilot CLI, leveraging the Model Context Protocol (MCP) and the specialized capabilities of the E2B substrate.

Conceptual Foundations of the Agent Sandbox

The architectural necessity of an agent sandbox is predicated on three fundamental value propositions: isolation, scale, and agency.[1] Traditional AI assistants operate within a restricted context window, generating code that must be manually vetted and executed by a human developer. In contrast, an agent sandbox provides a dedicated workstation—a micro-virtualized environment—where an agent possesses its own filesystem, network stack, and execution runtime.[1]

Isolation is achieved through hardware-level kernel isolation, typically utilizing Firecracker microVMs, which provide the security of a virtual machine with the speed of a container.[4] This allows for the execution of untrusted code, the installation of arbitrary dependencies, and the performance of destructive system operations within a "destroyable" environment.[1] Scale is realized through the ability to provision these environments in parallel. If a task requires a complex refactor, an orchestrator can spawn N independent agent forks, each in its own sandbox, to attempt the implementation using different strategies—a pattern known as "Best of N".[1] Agency is the culmination of these factors, granting the model the autonomy to act as a true engineer by granting it full control over its workstation.[1]

Operational Pillar	Technical Mechanism	Strategic Impact

Isolation	Firecracker MicroVMs / gVisor	Zero-trust execution of agent-generated code [4]
Scale	Parallel Sandbox Provisioning	Increased engineering throughput via Best-of-N [1]
Agency	Full Filesystem & Network Access	Transition from "Chatbot" to "Autonomous Worker" [2]
Persistence	NVMe-backed state (Sprites)	Persistent context across complex multi-day tasks [4]

Claude Code: Hierarchical Coordination and Safety

Claude Code architecture is designed as a multi-layered system that bridges the Anthropic large language model (LLM) with the developer's local environment through extensible protocols and specialized agent teams.[6] At the heart of this system is the Model Context Protocol (MCP), which serves as the communication substrate for connecting Claude to external data sources and tools.[6]

Agent Teams and Sub-agent Orchestration

Claude Code distinguishes itself through its support for "Agent Teams," an experimental feature that allows a lead agent to delegate tasks to specialized teammates.[6] These teammates can be assigned specific toolsets or denied access to dangerous commands via disallowedTools flags.[6] For the unified starter kit, the lead agent acts as the orchestrator, while the teammates operate within isolated sandboxes provided by the substrate. This hierarchy ensures that the lead agent maintains a high-level view of the project architecture while the teammates handle the mechanical work of implementation and testing.[6]

Interaction with these teams is managed through different display modes, including tmux or iTerm2 split-panes, which allow the human developer to monitor multiple agents in real-time.[6] A critical component of this orchestration is the "delegate mode," which restricts the lead agent to coordination tools only, ensuring that all code modification is offloaded to the sandboxed teammates.[6]

Permission Framework and Managed MCP

The safety of Claude Code is governed by a robust permission framework that supports multiple modes, ranging from plan (read-only analysis) to bypassPermissions (unrestricted tool use).[6] For a sandbox-integrated starter kit, the bypassPermissions mode is essential within the isolated container to allow for autonomous execution, while the local host remains

protected by more restrictive modes like dontAsk.[6]

Managed MCP configurations allow organizations to enforce strict control over the agent's environment.[7] By defining a managed-mcp.json file at the system level, administrators can prevent users from adding unauthorized MCP servers, effectively creating a "walled garden" of vetted tools.[7] This is particularly relevant for the unified kit, which must ensure that all external tool calls are routed through secure, audited channels.

Permission Mode	Behavior	Recommended Context
default	Prompts for each tool on first use	Standard local development [6]
acceptEdits	Auto-approves all file modifications	Trusted refactoring sessions [6]
plan	Analysis only; no modifications	Initial codebase exploration [6]
delegate	Coordination only for team leads	Multi-agent orchestration [6]
bypassPermissions	Skips all security prompts	Isolated Sandbox environments [6]

Gemini CLI: Extension Logic and Cross-Platform Portability

Gemini CLI provides a highly modular extension system based on JSON manifests and JavaScript logic, emphasizing developer-defined behaviors and "personality" overrides.[9] Its architecture is optimized for connecting to external services via the Model Context Protocol and custom TOML-based commands.[9]

The Extension Manifest and Skill Porter

The Gemini extension format (gemini-extension.json) allows developers to define MCP servers, persistent instructions via GEMINI.md, and specialized sub-directories for commands and skills.[9] A pivotal tool in the unified starter kit's arsenal is the skill-porter, which automates the bidirectional conversion between Claude Code skills and Gemini CLI extensions.[11] This tool enables the "write once, deploy anywhere" philosophy by analyzing directory structures,

extracting metadata, and transforming whitelisted tool sets into Gemini's blacklist-centric exclusion patterns.[11]

For the unified kit, the skill-porter logic will be integrated to ensure that a single sandbox management skill—such as the agent-sandboxes skill—can be seamlessly loaded into both Claude and Gemini environments without manual reconfiguration.[11]

System Prompt Overrides and Interactive Shells

Gemini CLI permits a full replacement of its core instructions through the GEMINI_SYSTEM_MD environment variable.[9] This "firmware-level" customization is critical for enforcing the rigorous software development lifecycle (SDLC) required in an agentic workflow, such as the iterative loops found in the "Pickle Rick" extension.[9] Furthermore, Gemini CLI's support for interactive shells via node-pty allows agents to use terminal-based editors and interactive command-line tools, bridging the gap between autonomous script execution and human-like terminal interaction.[10]

MCP Transport and Discovery

Gemini CLI supports three primary MCP transport types: Stdio (standard input/output subprocesses), SSE (Server-Sent Events), and Streamable HTTP.[9] For remote sandbox integration, the Streamable HTTP transport is preferred, as it allows for secure, token-authenticated connections to cloud-based sandbox gateways.[10] The discovery layer in Gemini CLI automatically sanitizes tool names and parameter schemas, ensuring compatibility even when multiple servers expose tools with conflicting names through automatic prefixing (e.g., serverName__toolName).[10]

GitHub Copilot CLI: Contextual Memory and Lifecycle Hooks

GitHub Copilot CLI represents the integration of agentic capabilities into the broader GitHub ecosystem, leveraging persistent repository memory and a powerful hook-based extensibility model.[14]

Copilot Memory and Agentic Identity

A distinguishing feature of the Copilot CLI is "Copilot Memory," which allows the agent to build a persistent, repository-specific understanding of coding conventions, patterns, and preferences over time.[14] This memory is validated against the current codebase before use, ensuring that the agent's "understanding" does not drift from the reality of the repository.[15] In a sandbox context, this memory serves as the agent's "long-term knowledge," while the sandbox provides the "short-term workspace."

The Hooks Architecture

Extensibility in Copilot CLI is driven by lifecycle hooks, which are scripts executed at critical junctures of the agent's execution loop.[14]

Hook Event	Execution Point	Primary Use Case
sessionStart	Start or resumption of an agent session	Sandbox provisioning and env setup [14]
preToolUse	Immediately before a tool call (e.g., bash)	Security intercept and command validation [14]
postToolUse	Immediately after a tool finishes	Result logging and state synchronization [14]
userPromptSubmitted	After the user types a message	Context injection and prompt analysis [14]
sessionEnd	Completion or termination of a session	Sandbox teardown and resource cleanup [14]

The preToolUse hook is of particular importance for the unified starter kit, as it allows for the implementation of a "Proxy" pattern. Every command the agent attempts to run locally can be intercepted, analyzed, and redirected to the isolated sandbox, effectively creating a transparent bridge between the user's terminal and the remote compute substrate.[14]

Agent Skills and the SKILL.md Standard

Like Claude Code, Copilot CLI utilizes the SKILL.md format to define specialized instructions and resources.[14] These skills are stored in .github/skills for project-level scope or ~/.copilot/skills for personal, cross-project expertise.[14] The unified kit will treat the SKILL.md file as the primary source of truth for the agent's operational logic, defining the backslash commands (e.g., \sandbox, \plan-build-host-test) used to orchestrate the environment.[2]

The E2B Substrate: Micro-Virtualization for Agents

The implementation of agent sandboxes relies on specialized infrastructure providers that prioritize developer experience and hardware-level isolation. E2B is the leading substrate in this category, offering Firecracker-based microVMs that can be provisioned in milliseconds.[4]

Firecracker and Hardware Isolation

The firecracker microVM provides a unique balance of performance and security. By utilizing a minimalist virtual machine monitor, E2B sandboxes achieve cold starts of 150–200ms, making them viable for just-in-time tool execution.[4] This isolation ensures that even if an agent is prompt-injected and commanded to execute malicious binaries, the damage is restricted to an ephemeral VM that will be destroyed at the end of the session.[1]

The total execution latency L_{total} for an agent task in a sandbox can be expressed as:

$$L_{total} = L_{provision} + L_{mcp\_handshake} + \sum_{i=1}^{n}(T_{exec\_i} +$$

Where $L_{provision}$ is the cold start time, n is the number of commands, and $T_{network}$ is the latency of streaming output back to the CLI.[4] The efficiency of E2B's micro-virtualization minimizes $L_{provision}$, allowing the agent to maintain a high level of responsiveness.

Docker MCP Partnership and Tool Catalogs

A significant advantage of the E2B platform is its partnership with Docker, which includes direct access to the Docker MCP Catalog.[4] Every E2B sandbox can automatically connect to over 200 MCP tools from providers like GitHub, Notion, and Stripe.[4] This "battery-included" approach means that a developer using the unified starter kit does not need to manually configure separate MCP servers for common services; they are available natively within the sandbox environment.[17]

Designing the Unified CLI Starter Kit

To create a single CLI starter kit that integrates agent sandboxes across Claude, Gemini, and Copilot, we must implement a modular, abstraction-based architecture.

The sbx CLI: A Python-Based Management Layer

The core of the integration is the sbx CLI, a Python-based utility that interacts with the E2B SDK to manage the lifecycle of sandboxes.[2] This utility must expose four primary commands:

1. **init**: Provisions a fresh sandbox and returns a unique sandbox_id.
2. **exec**: Executes a shell command inside a specific sandbox and streams stdout/stderr.
3. **files**: Handles directory listing, file reading, and writing within the sandbox filesystem.
4. **browser**: Manages the lifecycle of a Playwright-based browser instance for E2E testing.

This sbx utility will be wrapped in an MCP server, allowing all three assistant CLIs to call its functions via a standardized tool-calling interface.[6]

The Universal Skill Directory Structure

The starter kit will employ a unified directory structure that satisfies the requirements of all three assistants simultaneously.

File/Directory	Purpose
SKILL.md	Core instructions and metadata for Claude/Copilot [2]
gemini-extension.json	Manifest file for Gemini CLI loading [9]
sandbox_cli/	The sbx Python source code and environment [2]
prompts/	Markdown-based templates for specialized workflows [2]
hooks.json	Event-triggered script definitions for Copilot [14]
.mcp.json	Project-scoped tool configurations for Claude [6]

Hook-Based Automation and Provisioning

The kit will use each CLI's respective hooks to automate the sandbox lifecycle.

- **Claude Code**: Uses a SessionStart hook in settings.json to run sbx init and set the E2B_SANDBOX_ID environment variable.[6]
- **Gemini CLI**: Uses a SessionStart hook to initialize the environment and inject the sandbox's HTTP URL into the model's context.[9]
- **GitHub Copilot**: Uses the sessionStart event in hooks.json to provision the environment and the preToolUse hook to redirect local bash calls to the sbx exec command.[14]

Technical Requirements for Implementation

To accomplish the integration, the following prerequisites and technical steps must be completed.

Infrastructure and Authentication

The starter kit requires authenticated access to the E2B dashboard and the respective LLM providers.

- **E2B API Key**: Necessary for the Python SDK to provision microVMs.[2]
- **Anthropic API Key**: Required for Claude Code's model interaction.[6]
- **Google Cloud / Gemini Credentials**: Required for the Gemini CLI authentication loop.[19]
- **GitHub Personal Access Token (PAT)**: Required for repository integration and pull request generation.[17]

Software Stack

The local development environment must be equipped with the following runtimes:

- **Node.js 18+**: The primary runtime for the Gemini and Claude CLI packages.[17]
- **Python 3.10+**: Required for the sbx CLI and the E2B SDK.[2]
- **UV**: A high-performance Python package manager used to manage the sandbox CLI's dependencies and execution.[2]
- **GitHub CLI (gh)**: Essential for the "PR-generation" workflow, allowing agents to submit their changes for review directly from the terminal.[6]

Integrated Workflows: From Prompt to PR

The unified kit will implement the high-level workflows demonstrated in the "Reddit fix" example.[1]

1. **Ingestion**: The agent reads external context (e.g., a bug report or a Reddit thread) using the web_fetch tool.[1]
2. **Planning**: Using the \plan command, the agent generates a step-by-step implementation guide stored as a Markdown file.[2]
3. **Provisioning**: The agent calls sbx init to create a clean environment.[2]
4. **Execution**: The agent implements the code changes within the sandbox using sbx exec and verifies them with sbx browser.[2]
5. **Validation**: Automated tests are run to ensure the fix does not introduce regressions.[2]
6. **Submission**: Once verified, the agent uses the gh tool to create a branch, commit the changes, and open a Pull Request.[6]

Security and Governance in Agentic Workflows

As agents transition from passive assistants to active workers, the security of the execution environment becomes paramount.

Managed MCP and Locked-Down Modes

In enterprise environments, the unified kit must support "locked-down" modes to prevent unauthorized tool access.[7] Claude Code's managed-mcp.json and Gemini's mcp.allowed list provide the mechanisms for this control.[7] By whitelisting only the sbx MCP server and a few vetted external data sources (e.g., internal documentation), organizations can significantly reduce the risk of prompt injection-based data exfiltration.[7]

Network Sandboxing and Egress Filtering

While filesystem isolation is a strong foundation, network sandboxing is equally critical. Agents operating within a sandbox should be restricted by egress filtering and domain allowlists.[4] This prevents an agent from sending sensitive repository data to an attacker-controlled server if it is prompt-injected.[22] E2B and Modal both offer granular egress policies via CIDR allowlists, which should be configured by default in the starter kit.[4]

Identity and Auditing

Agents should be treated as distinct identities with their own API keys and logging trails.[3] Every tool call, shell command, and network fetch made by an agent should be recorded for auditing purposes.[3] The CC Usage and ccflare tools provide dashboards for monitoring Claude Code's token consumption and cost, which can be extended to track sandbox usage and execution logs.[24]

Scaling Engineering with Best-of-N Patterns

The ultimate power of the agent sandbox lies in its ability to scale engineering impact through parallel execution.

The Mathematics of Competitive Implementation

If $P(Success)$ is the probability that a single agent successfully solves a complex coding problem, the probability of at least one success among N parallel agents can be modeled as:

$$P(Success_{overall}) = 1 - (1 - P(Success))^N$$

As N increases, the likelihood of a high-quality implementation rises significantly, even if individual model performance is modest.[1] The unified starter kit facilitates this by allowing an orchestrator to spin up a fleet of sandboxes, assign the same task to each, and then use a "judge" agent or an automated test suite to select the best output.

Asynchronous Remote Tasks

By offloading long-running tasks to the cloud, developers can maintain their local focus while agents work in the background.[6] Claude Code's & prefix allows for the creation of independent web sessions that run asynchronously.[6] The /tasks command provides a centralized view of all background work, which can then be "teleported" back to the local machine once complete.[6]

Feature	Local Session	Web Session (Asynchronous)
Persistence	Tied to local terminal	Persistent in cloud infrastructure [6]
Parallelism	Serial execution	Concurrent execution of multiple tasks [6]
Resources	Uses local CPU/RAM	Uses isolated cloud VMs [6]
Access	Local filesystem only	Remote repo access via GitHub App [6]

Conclusion and Strategic Roadmap

The integration of agent sandboxes into a unified CLI starter kit for Claude, Gemini, and Copilot represents a critical milestone in the development of autonomous engineering systems. To achieve this, we must synthesize the hierarchical orchestration of Claude Code, the modular extension logic of Gemini CLI, and the hook-based automation of GitHub Copilot.

The roadmap for this implementation includes:

1. **Developing the Core sbx MCP Server**: A universal tool for sandbox management built in Python and compatible with all three assistant discovery layers.[2]
2. **Implementing a Cross-Platform Skill Template**: Utilizing the SKILL.md standard and the skill-porter utility to maintain a single source of operational truth.[2]
3. **Automating Provisioning via Lifecycle Hooks**: Leveraging sessionStart and initialization hooks to ensure a seamless, just-in-time compute substrate.[6]
4. **Establishing a Robust Security Layer**: Integrating managed MCP configs, egress filtering, and the Ramparts vulnerability scanner to protect the developer and the organization.[7]
5. **Exposing Scaling Patterns**: Providing built-in support for "Best-of-N" and

asynchronous remote tasks to maximize engineering throughput.[1]

As the agent runtime becomes the primary interface for software development, the ability to manage a fleet of isolated, autonomous workers will define the competitive advantage of modern engineering teams. This unified starter kit provides the blueprint for that future, enabling developers to scale their impact from a single terminal to a global swarm of AI agents.

Works cited

1. E2B Agent Sandboxes: The Space to Place your Claude Agents, accessed February 13, 2026, https://www.youtube.com/watch?v=1ECn5zrVUB4
2. disler/agent-sandbox-skill: An agent skill for managing ... - GitHub, accessed February 13, 2026, https://github.com/disler/agent-sandbox-skill
3. MCPs, Claude Code, Codex, Moltbot (Clawdbot) — and the 2026 Workflow Shift in AI Development - DEV Community, accessed February 13, 2026, https://dev.to/austinwdigital/mcps-claude-code-codex-moltbot-clawdbot-and-the-2026-workflow-shift-in-ai-development-1o04
4. sandbox-environments.md - how-to-ralph-wiggum - GitHub, accessed February 13, 2026, https://github.com/ghuntley/how-to-ralph-wiggum/blob/main/references/sandbox-environments.md
5. dtormoen/tsk: Task manager and sandbox for coding agents - GitHub, accessed February 13, 2026, https://github.com/dtormoen/tsk
6. CLI reference - Claude Code Docs, accessed February 13, 2026, https://code.claude.com/docs/en/cli-reference
7. Connect Claude Code to tools via MCP - Claude Code Docs, accessed February 13, 2026, https://code.claude.com/docs/en/mcp
8. Awesome Claude Code - Visual Directory, accessed February 13, 2026, https://awesomeclaude.ai/awesome-claude-code
9. Gemini CLI documentation | Gemini CLI, accessed February 13, 2026, https://geminicli.com/docs/
10. MCP servers with the Gemini CLI, accessed February 13, 2026, https://geminicli.com/docs/tools/mcp-server/
11. skill-porter/SKILL.md at main · AcidicSoil/skill-porter - GitHub, accessed February 13, 2026, https://github.com/AcidicSoil/skill-porter/blob/main/SKILL.md
12. AcidicSoil/skill-porter - GitHub, accessed February 13, 2026, https://github.com/AcidicSoil/skill-porter
13. Browse Extensions - Gemini CLI, accessed February 13, 2026, https://geminicli.com/extensions/
14. About GitHub Copilot CLI - GitHub Docs, accessed February 13, 2026, https://docs.github.com/en/copilot/concepts/agents/about-copilot-cli
15. About agentic memory for GitHub Copilot - GitHub Docs, accessed February 13, 2026, https://docs.github.com/en/copilot/concepts/agents/copilot-memory
16. Using hooks with GitHub Copilot CLI - GitHub Docs, accessed February 13, 2026,

https://docs.github.com/en/copilot/how-tos/copilot-cli/use-hooks
17. E2B sandboxes - Docker Docs, accessed February 13, 2026, https://docs.docker.com/ai/mcp-catalog-and-toolkit/e2b-sandboxes/
18. Claude Code - Documentation - E2B, accessed February 13, 2026, https://e2b.dev/docs/template/examples/claude-code
19. How to Build an MCP Server with Gemini CLI and Go | Google Codelabs, accessed February 13, 2026, https://codelabs.developers.google.com/cloud-gemini-cli-mcp-go
20. Set Up Gemini CLI for MCP: GitHub MCP Server - Docker, accessed February 13, 2026, https://www.docker.com/blog/how-to-set-up-gemini-cli-with-mcp-toolkit/
21. Guide to Integrating MCP Openapi with Gemini CLI, accessed February 13, 2026, https://openapi.com/mcp/gemini
22. NanoClaw solves one of OpenClaw's biggest security issues | Hacker News, accessed February 13, 2026, https://news.ycombinator.com/item?id=46976845
23. ramparts 0.4.0 - Docs.rs, accessed February 13, 2026, https://docs.rs/ramparts/0.4.0
24. A curated list of awesome skills, hooks, slash-commands, agent orchestrators, applications, and plugins for Claude Code by Anthropic - GitHub, accessed February 13, 2026, https://github.com/hesreallyhim/awesome-claude-code