

Architecting Production-Grade Agentic Coding Fleets: Isolation, Integration, and Delivery

1. Introduction and Central Research Question

The transition from localized AI coding assistants to autonomous, repository-scoped agentic fleets represents a fundamental evolution in software engineering. A production-grade coding-agent fleet is a highly orchestrated, distributed system capable of navigating, modifying, testing, and delivering software within the strict operational boundaries of real-world repositories. The central research question governing this analysis is: How should a production-grade coding-agent fleet be designed so that autonomous or semi-autonomous agents can safely work inside one or more real software repositories while maintaining Git integrity, test quality, security, traceability, reviewability, and human control?

The architecture must support the complete lifecycle of a software-development task—from task intake, repository discovery, and risk classification, through isolated environment provisioning, implementation, and rigorous independent testing, culminating in CI validation, human approval, and post-merge monitoring. At its core, the system must allow multiple agents to operate concurrently without corrupting the primary repository, overwriting parallel efforts, exposing secrets, or introducing untested regressions.

2. Foundations of Agentic Software Engineering

To design a robust architecture, precise distinctions must be drawn between varying levels of AI software-development capabilities. The fundamental difference between generating code in a chat interface and managing a complete software-delivery workflow lies in state management, environmental isolation, and continuous verification.

Capability Level	Characteristics	Boundary and Scope
Coding Assistant	Localized tool (e.g., chat or IDE plugin).	Relies on human context gathering. Executes isolated code generation without repository-wide architectural awareness.
Coding Agent	Executes iterative tasks via tool calling.	Can edit local files and execute commands but requires human

		orchestration for task decomposition and deployment.
Repository Agent	Read/write access to a specific repository.	Navigates directory structures and understands localized architecture. May lack full CI/CD integration capabilities.
Autonomous Agent	Handles task intake through to PR creation.	Minimal human intervention required. Operates within bounded policy gates and isolated execution environments.
Agent Fleet / Swarm	A coordinated network of specialized agents.	Governed by an <i>Orchestrator</i> . Agents work concurrently across a codebase assuming specialized roles (Planner, Implementer, Reviewer).

A chat model relies on a stateless prompt; a reliable repository-level engineering agent requires a persistent execution environment, a strictly managed Git worktree, a baseline understanding of existing continuous integration (CI) states, and the ability to dynamically route around failures. An *Orchestrator* serves as the deterministic workflow engine coordinating the fleet, managing the repository harness, and enforcing state transitions.¹

3. Repository Constraints and Bounded Execution

Agents must be strictly constrained to their authorized scopes. The runtime must prevent an agent from leaving the repository root, editing unrelated projects, reading arbitrary files from the host, accessing private SSH keys, modifying global Git configurations, or running unapproved network operations.

To enforce this, the orchestrator utilizes a repository policy manifest. This machine-readable file detects the repository root and integrates with native rules such as CODEOWNERS to define exactly where an agent may operate. The policy defines writable paths, read-only paths, forbidden paths (e.g., infrastructure-as-code directories, secret-file exclusions, vendor directories, and migration-directory boundaries), permitted commands, and network permissions.

3.1 Example Repository Policy Manifest

The following YAML structure represents a comprehensive policy manifest (.agent-policy.yml) governing fleet behavior within a specific repository:

YAML

```
version: "1.2"
repository: "org/core-backend"
permissions:
  writable_paths:
    - "src/api/**"
    - "tests/api/**"
  read_only_paths:
    - "**/*"
  forbidden_paths:
    - ".github/workflows/**"
    - "infrastructure/terraform/**"
    - "secrets.json"
    - "vendor/**"
  allowed_commands:
    - "npm run test"
    - "npm run lint"
    - "git commit"
    - "git push"
  network:
    egress_allowed: false
    allowed_domains:
merge_restrictions:
  requires_human_approval: true
  required_checks: ["security-scan", "unit-tests", "integration-tests"]
```

By enforcing this manifest, the orchestrator guarantees that an agent assigned to update a UI component cannot inadvertently modify cloud credentials or rewrite binary files.

4. Git Isolation: Branches vs. Git Worktrees

Concurrent agent execution demands robust state isolation. Providing multiple autonomous agents unrestricted access to a single, shared working directory is an anti-pattern that guarantees build contamination, file locks, and state collisions. Multiple agents checking out different branches within the same directory will inevitably overwrite uncommitted changes, while commands executed by one agent will interfere with the background processes of

another.

4.1 Isolation Mechanisms Comparison

Isolation Strategy	Filesystem State	Object Database	Cost / Complexity	Suitability for Concurrent Agents
Shared Repository (Branch Switching)	Shared	Shared	Low	Unsafe. Agents switching branches will collide, leaving uncommitted files and cross-contaminating build artifacts.
Separate Repository Clones	Fully Isolated	Independent	High	Suboptimal. Requires heavy synchronization, excessive disk usage, and redundant network fetching for each task.
Git Worktrees	Fully Isolated	Shared	Minimal	Optimal. Multiple working directories map to a single Git object store, enabling efficient, parallelized execution. ¹
Containerized	Fully Isolated	Independent	High	Situational.

Clones				Best for zero-trust cloud orchestration where ephemeral network and compute isolation is paramount per task.
--------	--	--	--	--

Git worktrees serve as the foundational primitive for parallel agentic development.² A worktree allows an agent to check out a dedicated branch into a unique filesystem directory while sharing the .git repository metadata. A decision matrix dictates that a branch alone is sufficient for single-human linear tracking; full clones are reserved for heavily containerized distributed processing; and Git worktrees are the optimal default for concurrent fleet operations on a shared host or persistent volume.¹

4.2 The "One Worktree Per Agent" Paradigm

Assigning one isolated Git worktree per active implementation agent eliminates filesystem race conditions. Worktrees must adhere to a deterministic naming convention. A recommended structure is:

- *Branch Name:* agent/<task-id>-<role>-<instance> (e.g., feature/TASK-241-auth-refresh-agent-03)
- *Worktree Path:* /var/agents/worktrees/<repo-id>/<task-id>-<role>-<instance>

Strict rules must govern this paradigm: there must be exactly one writable branch per implementation agent; no two agents may check out the same branch; direct work on the primary branch is strictly forbidden; and force-pushing shared branches requires explicit orchestrator override. Shared Git hooks, Git LFS objects, submodules, and sparse-checkout configurations are centrally managed by the shared Git object database, preventing redundant network operations. Read-only agents (e.g., repository explorers) do not require independent worktrees; they can safely inspect a shared, read-only snapshot or an indexed representation of the codebase.

5. Worktree Lifecycle and Task Orchestration

The full lifecycle of a worktree must be programmatically managed by a Worktree Registry. This registry tracks the repository ID, task ID, agent ID, branch name, base commit, worktree path, status, timestamps, and cleanup status.

The lifecycle sequence includes: validating the repository state, fetching the latest remote state, selecting the base commit, creating the task branch, creating the worktree, provisioning dependencies, running baseline tests, assigning the agent, monitoring changes, committing

work, pushing the branch, opening a pull request, and eventually archiving logs and pruning the worktree.

If a worktree creation fails, the task transitions to a blocked state. If the agent crashes, the orchestrator detects a missed heartbeat, commits the current patch snapshot, archives the state, and queues the worktree for cleanup. If a pull request is closed without merging, the corresponding worktree is marked as stale and pruned.

5.1 Example Worktree Registry Record

```
JSON
{
  "repo_id": "org/core-api",
  "task_id": "TASK-241",
  "agent_id": "impl-agent-03",
  "branch": "feature/TASK-241-auth",
  "base_commit": "a1b2c3d4",
  "worktree_path": "/worktrees/org-core-api/TASK-241",
  "status": "active",
  "created_at": "2026-07-25T01:00:00Z",
  "last_heartbeat": "2026-07-25T02:15:00Z",
  "pull_request_url": null,
  "cleanup_status": "pending"
}
```

6. Task Decomposition and Dependency-Aware Planning

To prevent chaotic implementations, software tasks must be systematically divided among agents. Decomposition by individual file is often too narrow because modern software changes require cross-module behavioral updates, interface updates, schema changes, and shared type modifications. Assigning work by feature, domain, or architectural layer provides agents with context completeness.

6.1 Dependency Graphs and Pre-Edit Declarations

The orchestrator must model a software change as a Directed Acyclic Graph (DAG). The DAG sequences tasks: Architecture tasks (defining interfaces) block Implementation tasks, which in turn block Test and Documentation tasks. For a feature spanning a backend API, database schema, and frontend, the schema migration must be prioritized.

To prevent edit collisions, the system utilizes pre-edit declarations and Change Manifests.

High-collision areas—such as dependency manifests (e.g., package.json), lock files, central routing files, and global configurations—require careful coordination. If two agents request overlapping write scopes, the orchestrator may serialize the tasks, reassign ownership, or enforce an interface contract that must be integrated prior to parallel development.

6.2 Example Change Manifest

The Change Manifest is created before implementation begins. The runtime continuously compares actual filesystem modifications against this manifest to detect unexpected file edits, new dependencies, or unplanned migrations.

```
JSON
{
  "task_id": "TASK-241",
  "objective": "Add role-based access control to export endpoint",
  "repository": "org/core-backend",
  "base_commit": "a1b2c3d4",
  "assigned_agent": "impl-agent-03",
  "writable_paths": ["src/api/export.ts", "tests/api/export.test.ts"],
  "read_only_paths": ["src/models/user.ts"],
  "expected_interfaces":,
  "test_commands": ["npm run test:api"],
  "risk_level": "moderate",
  "human_approval_requirements":,
  "rollback_strategy": "revert-pr"
}
```

7. Agent Roles in a Coding Fleet

A robust coding fleet separates responsibilities to preserve independent verification, preventing a single model from marking its own flawed logic as correct.

Agent Role	Responsibilities and Constraints
Repository Exploration Agent	Read-only. Inspects structure, locates relevant files, reads architecture documentation, and maps test commands. Utilizes embedding indexes and repository maps.

Planning Agent	Translates requests into technical scope. Identifies affected modules, defines acceptance criteria, decomposes tasks, and produces the Change Manifest.
Implementation Agent	Modifies assigned code within an isolated worktree. Follows conventions, runs scoped validation, and commits atomic changes.
Testing Agent	Operates independently of the Implementer. Writes independent tests against acceptance criteria, reproduces bugs, searches for edge cases, and reports regressions.
Code-Review Agent	Inspects diffs for correctness, maintainability, naming, and error handling. Produces structured, actionable review comments rather than low-value suggestions.
Security-Review Agent	Identifies vulnerabilities (e.g., injection, path traversal). Inspects authentication logic, traces untrusted data, and reviews dependency changes.
Documentation Agent	Updates README.md, API schemas, and changelogs. Verifies that documented examples compile and that API responses match the updated implementation.
Integration Agent	Combines compatible branches, resolves integration issues, runs broader test suites, and prepares the final pull request.
Release Agent	Validates release requirements, verifies versioning, checks deployment readiness, and monitors post-merge telemetry.

7.1 Read-Only Exploration and Repository Memory

Read-only exploration agents drastically reduce risk while performing repository mapping, dependency analysis, and bug localization. They operate via read-only filesystem mounts or Git index protection, with no package installation or network write capabilities.

To supply agents with relevant context without overwhelming their context windows (a phenomenon known as "context rot" ⁵), fleets utilize the **Model Context Protocol (MCP)**.⁶ MCP provides a standardized interface for agents to query abstract syntax trees, language-server indexes, and call graphs.⁸ Instead of loading an entire repository, the orchestrator provisions a concise "context pack" containing the task description, relevant files, related symbols, and recent commits.

7.2 Example Task Plan

```
JSON
{
  "task_id": "TASK-241",
  "objective": "Implement RBAC for exports",
  "dag": [
    {"step": 1, "action": "Explore", "agent": "explorer-01"},
    {"step": 2, "action": "Plan", "agent": "planner-01"},
    {"step": 3, "action": "Implement", "agent": "impl-agent-03", "depends_on": },
    {"step": 4, "action": "Verify", "agent": "test-agent-02", "depends_on": },
    {"step": 5, "action": "Review", "agent": "sec-agent-01", "depends_on": }
  ],
  "acceptance_criteria":
}
```

8. Implementation, Testing, and Verification

8.1 Test-Driven Development (TDD) and Baseline Validation

Agents should follow test-driven development cycles: understanding required behavior, writing a failing test, confirming the expected failure, implementing the change, and running broader regression tests. Crucially, the orchestrator enforces **Baseline Validation** before any code is changed. The baseline captures existing test status, lint errors, and flaky tests, ensuring that pre-existing environmental failures are distinguished from agent-induced regressions.

Relying on the same agent to write tests and implementation code introduces significant risk, including overfitting (tests that reproduce the implementation rather than requirements) and

weak assertions. Therefore, an independent Testing Agent operates in a clean cloned environment (or an ephemeral preview environment) to verify the implementation against the original acceptance criteria.

8.2 Security and Code Review Agents

Review agents analyze diffs iteratively. To prevent AI agents from generating hundreds of pedantic, non-blocking comments, reviews must output a structured schema defining severity, confidence, and recommended corrections. Security agents perform source-to-sink analysis, flagging unsafe command execution or tenant-isolation failures. Security findings must be evidence-based and calibrated; automatically classifying every theoretical issue as critical results in alert fatigue and blocked workflows.

8.3 Example Structured Review Finding

```
JSON
{
  "comment_id": "REV-992",
  "reviewer": "sec-agent-01",
  "severity": "blocking",
  "file": "src/api/export.ts",
  "line": 42,
  "finding": "Insecure authorization check.",
  "evidence": "Endpoint checks user.isLoggedIn but fails to verify user.role === 'admin'.",
  "consequence": "Any authenticated user can trigger an administrative data export, leading to data exfiltration.",
  "recommended_correction": "Add authorization middleware: `app.use('/export', requireAdmin)`.",
  "confidence": 0.99,
  "blocking_status": true
}
```

9. Software Delivery, Integration, and Conflict Resolution

9.1 CI/CD Integration and Automated Pull Requests

Coding agents interact with CI/CD systems (e.g., GitHub Actions, GitLab CI) by reading workflow definitions, observing check statuses, retrieving logs, and classifying failures. Modifying CI/CD configuration files is strictly controlled, requiring elevated review, as such

changes can alter security checks, build provenance, or secrets access.

Automated pull requests should default to a "draft" state until autonomous testing passes. A PR must include a clear title, problem statement, scope, files changed, generated tests, known limitations, agent identities, and a rollback plan.

9.2 Example Pull Request Description

Problem Statement

The administrative export endpoint lacks Role-Based Access Control (RBAC), allowing any authenticated user to export data.

Solution Summary

Implemented RBAC middleware on /api/export requiring the admin role. Non-admin requests now return 403 Forbidden.

Scope and Files Changed

- src/api/export.ts (Modified: Added middleware)
- tests/api/export.test.ts (Added: 3 new test cases)

Tests Added and Run

- test_admin_export_success: PASS
- test_user_export_forbidden: PASS
- test_unauthenticated_export_unauthorized: PASS

Security Considerations

Validates JWT roles explicitly. No new dependencies added.

Rollback Information

Revert this PR. No database schema changes are associated with this feature.

Generated By: impl-agent-03 (Anthropic Claude 3.5 Sonnet) **Reviewed By:** sec-agent-01, test-agent-02

9.3 Revision Loops and Merge Conflict Handling

During the review-revision lifecycle, findings are classified, the implementation agent revises the worktree, tests are rerun, and review threads are addressed. The system utilizes comment-tracking IDs and maximum iteration thresholds (e.g., max 3 revisions) to prevent infinite loops caused by contradictory agent feedback. Escalation rules route stagnant loops to human arbitration.

Merge conflicts represent a significant challenge for parallel fleets. While textual conflicts (e.g., Git conflict markers) are common, *semantic conflicts* are more dangerous (e.g., Agent A

renames a function that Agent B calls in a separate branch, resulting in a textually clean merge that fails compilation). AI semantic conflict resolution—utilizing tools like Git Merger AI⁹ or Verdent workspaces¹⁰—analyzes the behavioral intent of both branches to synthesize a functional integration.¹¹ All resolutions require strict post-merge validation (compilation, type checks, tests) by the Integration Agent.

9.4 Integration Branches and Trunk-Based Development

For fast, AI-assisted shipping, Trunk-Based Development (TBD) with short-lived feature branches is the most effective model.¹² To maintain an always-releasable primary branch, AI agents integrate changes frequently behind *feature flags*.¹² Risky changes are isolated via flags rather than languishing in long-lived feature branches. Merge queues are employed to batch-test PRs, preventing race conditions where multiple PRs pass CI individually but break the build when merged sequentially.

9.5 Example Merge Conflict Resolution Record

```
JSON
{
  "conflict_id": "MC-105",
  "base_branch": "main",
  "target_branch": "feature/TASK-241-auth",
  "conflicted_file": "src/api/routes.ts",
  "conflict_type": "semantic",
  "description": "Agent A added the export route; Agent B refactored the router instantiation.",
  "resolution_strategy": "ai-semantic-merge",
  "integration_agent": "integ-agent-01",
  "post_merge_validation": {
    "compilation": "pass",
    "tests": "pass"
  }
}
```

10. Dependency Management and External Packages

Agents frequently attempt to install arbitrary packages to simplify implementation. Dependency management requires strict governance: any modification to package.json or lock files must include justification, alternative assessments, version selection, and a security review for typosquatting or abandoned packages. Minor and major upgrades require explicit human approval, while patch upgrades for development dependencies may be automated if integration tests pass. The fleet must record all dependency changes and ensure lock files are

consistently updated to prevent environment drift.

11. Security, Sandboxing, and Environment Isolation

11.1 Secrets and Zero Trust

Agents must operate without unrestricted access to secrets. Following Zero Trust Architecture (ZTA) principles ⁵, agents are granted short-lived, scoped credentials (e.g., OIDC workload identity federation).⁵ Long-lived credentials and shared personal tokens are strictly prohibited. Secrets are injected securely into the environment execution variables and never included in prompts. All command outputs and telemetry logs are aggressively redacted to prevent token exfiltration.

11.2 Sandboxes and Containers

The Thoughtworks Technology Radar highly recommends sandboxed execution for coding agents as a "sensible default".¹³ Executing code directly on a host machine exposes it to catastrophic risks, including lateral network movement and accidental deletion of unversioned files.

Sandboxing Technology	Filesystem Isolation	Resource Limits	Startup Time	Statefulness	Best Use Case
Host Execution	None	None	Instant	Stateful	Prohibited. High risk of system compromise.
OS-Level Sandboxes	High (Namespace/ACL)	Low	Fast	Stateful	Local agent operation (e.g., sandbox-exec on macOS ¹ , Bubblewrap on Linux ¹⁴).
Docker Containers	High	High	Moderate	Ephemeral	Standard CI/CD

					execution and isolated testing.
MicroVMs (e.g., Shuru)	Extreme	Strict	Fast	Ephemeral	Zero-trust execution of untrusted, high-risk code. ⁵
Persistent Sandboxes (e.g., Sprites)	High	High	Moderate	Stateful	Session recovery for long-running autonomous tasks. ¹⁴

The optimal architecture utilizes a layered model: a Git worktree provides source isolation, a container or microVM (like Shuru) provides execution isolation¹, scoped credentials govern service access, and resource quotas ensure operational safety.¹⁴

11.3 Command Execution Policies

Command execution is heavily gated. String-based filtering is insufficient due to shell expansion, aliases, and subshells. Execution must be validated at the sandbox layer.

- *Low Risk*: Read files, search text, inspect Git status, run linters. (Auto-approved).
- *Medium Risk*: Install dependencies, generate code, start local services. (Requires sandbox isolation).
- *High Risk*: Alter Git remotes, modify CI configuration, access external networks. (Requires orchestrator approval).
- *Critical Risk*: Deploy to production, modify production data, rotate secrets, force-push branches. (Requires human execution/approval).

11.4 Example Command Policy

```
JSON
{
  "policy_id": "CMD-POL-01",
  "risk_level": "medium",
```

```

"allowlist": [
  "^npm install.*",
  "^npm run test.*",
  "^git (add|commit|status|diff).*"
],
"denylist": [
  "^rm -rf.*",
  "^curl.*",
  "^wget.*"
],
"shell_expansion_allowed": false
}

```

11.5 Untrusted Repository Content

Coding agents must treat all repository content as potentially untrusted data. Malicious actors can embed prompt-injection payloads inside README.md files, issues, or generated test data, instructing the agent to exfiltrate credentials or disable security policies. The system must enforce a trust hierarchy: runtime controls and system policy manifests strictly override any textual instructions discovered inside the repository.

12. Context, Memory, and Monorepo Workflows

12.1 Environment Reproducibility

Agent environments must be perfectly reproducible to avoid false test failures caused by "environment drift." The system must capture the operating system, container image, runtime versions, lock files, environment variables, and tool versions (e.g., using Nix, DevContainers, or strict Dockerfiles).

12.2 Monorepos vs. Polyrepos

Agentic coding fleets encounter severe friction in multi-repository (polyrepo) environments due to fragmented context, lack of atomic cross-project writes, and obscured dependency graphs.¹⁵ When a change spans a backend API repository and a frontend UI repository, the orchestrator must manually construct cross-repository task graphs, coordinate pull requests, pin versions, and manage release ordering.

Monorepos inherently solve the AI visibility problem.¹⁷ With a unified file system, an agent can trace a user action from the frontend UI layer, through the API, into the database schema—all within a single context boundary.¹⁶ Build systems like Nx, Bazel, and Turborepo provide deterministic, graph-aware tooling.¹⁸ When an agent modifies a Go backend, Nx immediately calculates the affected dependency graph, allowing the agent to understand how the change impacts a downstream React application.¹⁸ For organizations unable to adopt monorepos, tools like Polygraph create a "synthetic monorepo," connecting distributed repositories into a single

logical dependency graph for the agent fleet.¹⁵

12.3 Example Multi-Repository Release Manifest

```
JSON
{
  "release_id": "REL-405",
  "objective": "Cross-repo schema update",
  "repositories": [
    {"repo": "org/core-api", "branch": "feature/schema-v2", "merge_order": 1},
    {"repo": "org/frontend-client", "branch": "feature/api-v2", "merge_order": 2}
  ],
  "compatibility_requirements": "Backward compatible API v1 remains active.",
  "rollback_strategy": "Revert frontend first, then API."
}
```

13. Observability, Failure Handling, and Orchestration

13.1 Failure Scenarios and Remediation

Production fleets must autonomously handle a vast array of failure modes. Checkpointing strategies (e.g., worktree archives, patch snapshots) ensure that progress is not lost upon a crash, without polluting permanent Git history with intermediate noise.

Failure Scenario	Prevention / Detection	Remediation / Ownership / Recovery
Two agents edit shared type file	Prevented by task DAG and file leases. Detected by Git index conflict.	Orchestrator serializes work; Integration Agent synthesizes changes.
Dependency conflict on lock file	Dependency changes flagged as High Risk. Detected via pre-edit manifest.	Rebasing target; central lock file regeneration; rerun CI.
Agent modifies primary branch	Prevented by Git hooks and orchestrator branch	Action blocked at runtime; system alert triggered.

	validation.	
Agent runs destructive shell command	Prevented by sandbox-exec/microVM isolation ¹ and command policy.	Command blocked; agent session terminated; worktree wiped.
Malicious prompt in README	Prevented by trust hierarchy (repo = data).	Egress firewall prevents exfiltration; system policy overrides instructions.
Test passes without asserting behavior	Detected by Testing Agent performing mutation testing.	Code-Review Agent blocks PR; task returned to Implementer.
Review agent raises invalid finding	Deduplication engine tracks comment IDs to prevent infinite loops.	Max iteration threshold hit; escalated to human arbitration.
CI fails due to flaky test	Detected via Baseline Validation run prior to implementation.	Flake ignored in diff evaluation; logged to technical debt tracker.
Semantic conflict post-CI	Detected by post-merge validation in Integration Branch.	AI semantic resolver proposes fix ² ; Integration Agent applies forward fix.
Agent exposes token in logs	Prevented by OS-level secret redaction before telemetry aggregation.	Key rotated via automated OIDC pipelines; session killed.
Vulnerable transitive package added	Detected by Security-Review Agent scanning lock files.	PR blocked; alternative package requested by Planner.
Worktree remains after crash	Heartbeat monitor in Worktree Registry detects timeout.	Stale worktree pruned; uncommitted diff saved to patch archive.
Agent edits outside	Detected by Change	Commit rejected; agent

declared scope	Manifest validation during Git commit.	forced to revert unauthorized files.
Irreversible database migration	High-risk classification mandates human approval.	Handled via deployment rollback and feature flags. ¹²
Multi-repo release mismatch	Prevented by Nx graph analysis ¹⁸ or Polygraph orchestrator. ¹⁵	Strict CI pipeline sequencing ensures backward compatibility.

13.2 Observability and Auditability

Comprehensive telemetry is required. The system must track task assignments, worktrees, files changed, commands run, test results, security findings, API costs, and human approvals. Correlation identifiers allow the organization to reconstruct the complete history of a change, tracing a merged PR back to the specific model version and prompt that generated the implementation.

13.3 Example Audit Event

```
JSON
{
  "event_id": "AUD-77382",
  "timestamp": "2026-07-25T02:15:00Z",
  "correlation_id": "TASK-241",
  "actor": "impl-agent-03",
  "action": "execute_command",
  "target": "npm install bcrypt",
  "status": "success",
  "sandbox_id": "microvm-shuru-09",
  "policy_applied": "CMD-POL-01"
}
```

13.4 State Machine and Fleet Orchestration

The orchestrator must be a deterministic workflow engine, not an LLM. Utilizing an LLM to govern state transitions introduces catastrophic non-determinism. LLMs are reserved for planning, implementation, and semantic review; the orchestrator handles permissions, Git operations, and quality gates deterministically.

Task State Machine:Received → Exploring → Planning → Awaiting Approval → Worktree Provisioning → Baseline Testing → Implementing → Testing → Reviewing → Revising (Loop) → CI Running → Ready for Human Review → Approved → Merge Queued → Merged → Deploying → Completed.
Failures trigger specific recovery paths (e.g., Testing Failed transitions back to Revising, while Command Timeout transitions to Blocked).

13.5 Example CI Failure Diagnosis

```
JSON
{
  "diagnosis_id": "DIAG-442",
  "task_id": "TASK-241",
  "ci_job": "Integration Tests",
  "failure_type": "assertion_error",
  "log_snippet": "Expected 403 Forbidden, received 200 OK at /api/export",
  "agent_analysis": "The RBAC middleware was applied to the router, but the specific route `/api/export` bypassed the middleware due to incorrect ordering in Express.",
  "recommended_action": "Route to impl-agent-03 to move `app.use(requireAdmin)` above the route definition."
}
```

14. Deployment Boundaries, Quality Gates, and Remediation

14.1 Quality Gates and Risk Classification

Mandatory quality gates control progression based on a risk-classification system. Risk signals include the number of files changed, core-module modifications, public API changes, database migrations, and dependency updates.

- *Routine Risk*: Scoped tests, linting, agent review, draft PR creation.
- *Moderate Risk*: Full relevant tests, independent code review, security checks, human merge approval.
- *High/Critical Risk*: Architecture approval, strict independent testing, staging validation, rollback plan evaluation, and human merge/deployment approval.

Human-in-the-loop controls must present concise approval experiences, showing the requested action, the diff, identified risks, test coverage, and the rollback plan. Meaningless

approval prompts that force humans to approve without context must be avoided to prevent alert fatigue.

14.2 Deployment Boundaries and Rollback

Coding agents may create preview environments and deploy to disposable test environments autonomously. However, staging deployments require policy approval, and production deployments demand explicit human authorization.

Rollback strategies for agent-generated changes include Git reverts, feature-flag disablement, and deployment rollbacks. Database migrations and irreversible external side effects require specialized forward-fix methodologies, as they cannot be reliably rolled back via standard Git operations. High-risk pull requests must strictly enforce the presence of a validated rollback plan prior to merge.

15. Technical Architecture and Platform Neutrality

15.1 Provider-Neutrality

The coding fleet must not depend on a single model provider or vendor. The architecture necessitates deep abstractions for repository reading, patch generation, command execution, planning, tool calling, and context assembly. The system must support dynamic routing between OpenAI, Anthropic, Google, and locally hosted open-source models based on role requirements. Assuming one model is superior for all roles (e.g., Exploration vs. Security Review) is an architectural flaw.

15.2 Practical Reference Architecture

A production implementation requires a highly decoupled component architecture:

- **Coding-Fleet Orchestrator:** The deterministic state machine.
- **Repository Registry & Policy Engine:** Enforces boundaries and manifests.
- **Task Scheduler & Worktree Manager:** Allocates worktrees and tracks the Git lifecycle.
- **Sandbox Manager:** Provisions layered execution environments (e.g., Docker, Shuru microVMs).⁵
- **Context & Indexing Service:** Serves MCP integrations for Progressive Context Disclosure.⁶
- **Agent Runtime:** The provider-neutral LLM execution layer.
- **Review, Test, and Security Services:** Independent validation engines.
- **Secrets Broker:** Manages OIDC federation and credential injection.

15.3 Minimum Viable vs. Advanced Fleets

An MVP suitable for immediate use includes repository registration, a basic policy file, a Read-Only Explorer, a Planner, one Implementation Agent, one Test/Review Agent, isolated worktree provisioning¹, local test execution, and draft PR creation.

An Advanced Coding Fleet scales this to dozens of concurrent agents across multiple organizations. It includes elastic microVM sandbox provisioning⁵, advanced semantic

merge-conflict detection ⁹, automated task-graph optimization, change-impact prediction, cross-repository release orchestration via tools like Polygraph ¹⁵, and autonomous continuous repository maintenance.

16. Repository Integrity Guarantees

To ensure ultimate safety, the coding fleet must continuously verify explicit architectural invariants:

1. The primary branch is never modified directly by an implementation agent.
2. Every writable agent is strictly assigned a unique branch and isolated worktree.
3. Every code modification is attributable to a specific task, agent run, and model version.
4. Every changed file must fall within the approved scope defined by the Change Manifest; deviations are hard-blocked.
5. Every merged change passes required quality gates, including independent testing and security review.
6. Every high-risk action requires human approval.
7. No production secret is available to routine coding agents.

17. Conclusion

What is the smallest safe architecture that allows several coding agents to work concurrently inside real repositories, while preserving enough isolation, testing, review, auditability, and human control to evolve later into a largely autonomous software-delivery fleet?

The minimum viable safe architecture fundamentally rejects shared-filesystem execution. It requires a deterministic, non-LLM orchestrator that provisions individual Git worktrees ¹ inside tightly scoped, layered sandboxes (utilizing OS-level restrictions or ephemeral microVMs).⁵ This isolated execution environment is paired with a strictly enforced, role-based fleet model that structurally separates Implementation Agents from Testing and Review Agents to eradicate confirmation bias.

Context must be managed not by dumping monolithic codebases into prompts, but by leveraging the Model Context Protocol (MCP) for targeted, progressive context disclosure ⁵, ideally within a graph-aware monorepo environment (such as Nx or Bazel).¹⁸ Safety is guaranteed through machine-readable Change Manifests that restrict modifications to predetermined scopes, preventing lateral architectural drift. Finally, all agentic delivery must be routed through standard Trunk-Based Development workflows utilizing feature flags.¹² This ensures no agent-generated code reaches production without passing Baseline Validation, automated security gates, and explicit human merge authorization. This architecture provides absolute Git integrity and failure isolation today, establishing a scalable, provider-neutral foundation capable of supporting the fully autonomous software delivery lifecycles of the future.

Works cited

1. Sandboxes and Worktrees: My secure Agentic AI Setup | Mike ..., accessed July

- 25, 2026,
<https://mikemcquaid.com/sandboxed-agent-worktrees-my-coding-and-ai-setup-in-2026/>
2. Git Worktrees for AI Coding: How to Run Multiple Agents Without ..., accessed July 25, 2026,
<https://www.mindstudio.ai/blog/git-worktrees-parallel-ai-coding-agents>
 3. Git Worktrees Tutorial #5 - Worktrees in Agentic Coding Workflows | daily.dev, accessed July 25, 2026,
<https://daily.dev/posts/git-worktrees-tutorial-5---worktrees-in-agentic-coding-workflows-pqh6kikzs>
 4. Agentic Coding: Git Worktrees and Agent Skills for Parallel Workflows - Tech Blog (mainly!), accessed July 25, 2026,
<https://blog.shanelee.name/2026/02/03/agentic-coding-git-worktrees-and-agent-skills-for-parallel-workflows/>
 5. Techniques | Thoughtworks Ecuador, accessed July 25, 2026,
<https://www.thoughtworks.com/en-ec/radar/techniques>
 6. Model Context Protocol: What Security Teams Need to Know Before Agents Scale - Zenity, accessed July 25, 2026,
<https://zenity.io/academy/model-context-protocol-explained>
 7. Model Context Protocol, accessed July 25, 2026,
<https://modelcontextprotocol.io/docs/getting-started/intro>
 8. What is the Model Context Protocol (MCP)? - Databricks, accessed July 25, 2026,
<https://www.databricks.com/blog/what-is-model-context-protocol>
 9. Merge Resolver: Automatic Git Conflict Resolution Using AI - DEV Community, accessed July 25, 2026,
<https://dev.to/roshanacharya/merge-resolver-automatic-git-conflict-resolution-using-ai-16nl>
 10. How Should You Review Merge Conflicts From Parallel AI Agents?, accessed July 25, 2026,
<https://www.verdent.ai/guides/answers/review-merge-conflicts-parallel-ai-agents>
 11. Resolve Git Merge Conflicts faster with Artificial Intelligence (AI) - DISCOVER, accessed July 25, 2026,
<https://www.arcadsoftware.com/discover/resources/blog/resolve-git-merge-conflicts-faster-with-artificial-intelligence-ai/>
 12. Agentic Git Workflow - DEV Community, accessed July 25, 2026,
https://dev.to/igor_grieder/agentic-git-workflow-high
 13. Sandboxed execution for coding agents | Technology Radar | Thoughtworks United States, accessed July 25, 2026,
<https://www.thoughtworks.com/en-us/radar/techniques/sandboxed-execution-for-coding-agents>
 14. Sandboxed execution for coding agents | Technology Radar - Thoughtworks, accessed July 25, 2026,
<https://www.thoughtworks.com/radar/techniques/sandboxed-execution-for-coding-agents>
 15. The Monorepo Advantage for AI Agents | daily.dev, accessed July 25, 2026,

- <https://daily.dev/posts/the-monorepo-advantage-for-ai-agents-cjytesn0r>
16. Monorepo vs Multi-Repo: Why AI Agents Tip the Scale - Francis Eytan Dortort, accessed July 25, 2026,
<https://dortort.com/posts/monorepo-vs-multi-repo-why-ai-agents-tip-the-scale/>
 17. Monorepo vs. Polyrepo for Multi-Stack Vibe Coding: A Developer's Decision Framework, accessed July 25, 2026,
<https://medium.com/@arohitu/monorepo-vs-polyrepo-for-multi-stack-vibe-coding-a-developers-decision-framework-06d535fb110e>
 18. Nx vs Turborepo, accessed July 25, 2026,
<https://nx.dev/docs/guides/comparisons/nx-vs-turborepo>