agit

Infrastructure-Aware Git Orchestration for AI Agents

Architecture & Implementation Specification

Version 0.1.0 | February 2026
Language: Go | License: MIT

1. Overview

1.1 Problem Statement

AI coding agents (Claude Code, Cursor, Codex, OpenClaw) interact with Git repositories daily, but they operate without persistent awareness of their infrastructure. Each agent session starts from scratch with no knowledge of what repositories are available, what other agents are doing, or what state the codebase is in. When multiple agents work on the same repository, there is no coordination layer to prevent conflicts, manage branches, or sequence work.

This creates three categories of pain:

- **Context loss:** Agents forget which repositories they have access to between sessions. A developer must re-explain the infrastructure every time.
- **Collision:** Multiple agents editing the same files in the same repo create merge conflicts discovered only at merge time, wasting work.
- **No coordination:** There is no mechanism for agents to claim tasks, signal what files they are touching, or sequence their work.

1.2 Solution

`agit` is a CLI tool and MCP server that provides AI agents with persistent, queryable awareness of their Git infrastructure. It manages a local registry of repositories, orchestrates Git worktrees for agent isolation, detects cross-worktree conflicts, and coordinates task assignment across multiple agents.

1.3 Design Principles

- **Local-first:** All state is stored locally. No cloud dependency. Works offline.
- **Git-native:** Uses standard Git primitives (worktrees, branches, diffs). No custom VCS. Works with any Git host.
- **Agent-agnostic:** Works with any MCP-compatible agent. Not tied to Claude, Cursor, or any specific tool.
- **Human-friendly:** The CLI is designed for developers first. Agents use the same tool via MCP.
- **Minimal:** Does one thing well. Not a dev environment, not a CI system, not a project manager.

1.4 Architecture Overview

The system consists of four components:

Component	Role	Interface
CLI (agit)	Human interface for managing repos, worktrees, tasks	Terminal commands
Registry (SQLite)	Persistent state store for repos, worktrees, agents, tasks	Internal API
Git Engine	Wrapper around Git operations (worktrees, branches, diffs, merges)	Internal API
MCP Server	Agent interface; exposes registry and operations as MCP tools/resources	MCP protocol (stdio/SSE)

2. Data Model

All persistent state lives in a SQLite database at `~/.agit/agit.db`. SQLite provides ACID transactions for safe concurrent access from multiple agents, and the entire state is a single portable file.

2.1 repos

Tracks all registered Git repositories.

Column	Type	Description
`id`	TEXT PK	UUID v4
`name`	TEXT UNIQUE	Human-readable alias (e.g. "openclaw")
`path`	TEXT	Absolute path to repo root
`remote_url`	TEXT	Origin remote URL

`default_branch`	TEXT	e.g. "main", "master"
`added_at`	TIMESTAMP	When repo was registered
`last_synced`	TIMESTAMP	Last git fetch timestamp
`metadata`	JSON	Extensible key-value store

2.2 worktrees

Tracks all agent worktrees created by agit.

Column	Type	Description
`id`	TEXT PK	UUID v4
`repo_id`	TEXT FK	References repos.id
`path`	TEXT	Absolute path to worktree directory
`branch`	TEXT	Branch name (auto-prefixed agit/)
`agent_id`	TEXT NULL	Which agent is working here
`task_description`	TEXT NULL	What the agent is doing
`status`	TEXT	active | completed | stale | conflict
`created_at`	TIMESTAMP	Creation time
`updated_at`	TIMESTAMP	Last status change

2.3 agents

Tracks agents that have connected via MCP or been registered manually.

Column	Type	Description
`id`	TEXT PK	Agent identifier (self-reported or auto-generated)
`name`	TEXT	Human-readable name (e.g. "claude-code-1")
`type`	TEXT	Agent type: claude-code, cursor, codex, openclaw, custom
`status`	TEXT	active | idle | disconnected
`current_worktree_id`	TEXT NULL	FK to worktrees.id
`last_seen`	TIMESTAMP	Last heartbeat or action

2.4 tasks

Optional task coordination. Agents can create, claim, and complete tasks to avoid duplicating work.

Column	Type	Description
`id`	TEXT PK	UUID v4
`repo_id`	TEXT FK	Which repo this task relates to
`description`	TEXT	What needs to be done
`status`	TEXT	pending | claimed | in_progress | completed | failed
`assigned_agent_id`	TEXT NULL	FK to agents.id
`worktree_id`	TEXT NULL	FK to worktrees.id
`created_at`	TIMESTAMP	When task was created
`completed_at`	TIMESTAMP NULL	When task finished
`result`	TEXT NULL	Completion notes, PR URL, etc.

2.5 file_touches

Tracks which files each worktree has modified, enabling conflict detection without continuous file watching. Updated on-demand via git diff.

Column	Type	Description
`repo_id`	TEXT	FK to repos.id
`worktree_id`	TEXT	FK to worktrees.id
`file_path`	TEXT	Relative path from repo root
`change_type`	TEXT	added | modified | deleted | renamed
`updated_at`	TIMESTAMP	When this was last detected

3. Configuration

Configuration lives at `~/.agit/config.toml`. It defines server settings and default behaviors. Repos are registered via the CLI (stored in SQLite), not in the config file.

3.1 Config Schema

```
[server]
transport = "stdio"       # stdio (default) or sse
port = 3847               # port for SSE transport

[defaults]
branch_prefix = "agit/"   # prefix for auto-created branches
worktree_dir = ".worktrees" # where worktrees are created (relative to repo)
cleanup_stale_after = "24h" # auto-mark worktrees as stale
auto_conflict_check = true  # check conflicts before merge
```

```
[agent]
heartbeat_interval = "30s"  # how often agents should ping
stale_after = "5m"          # mark agent disconnected after silence
```

3.2 Per-Repo Config (Optional)

A `.agit.toml` file can be placed in any repo root to override defaults for that repository.

```
[worktree]
dir = ".agit-worktrees"   # custom worktree location
branch_prefix = "agent/"  # override branch prefix

[tasks]
auto_create_from = "issues" # future: auto-import from GitHub issues
```

4. CLI Interface

All commands use the `agit` prefix. The CLI is built with Cobra and follows standard Unix conventions.

4.1 agit init

Initialize agit. Creates `~/.agit/` directory, SQLite database, and default config.

```
$ agit init
Created ~/.agit/config.toml
Created ~/.agit/agit.db
agit initialized. Add repos with: agit add <path>
```

4.2 agit add <path> [--name alias]

Register a Git repository with agit. Auto-detects remote URL and default branch.

```
$ agit add ~/projects/openclaw --name openclaw
Registered: openclaw
  Path:   /home/user/projects/openclaw
  Remote: git@github.com:openclaw/openclaw.git
  Branch: main
```

Behavior: Validates the path is a Git repo, reads origin remote, detects default branch, inserts into repos table.

4.3 agit repos

List all registered repositories and their current state.

```
$ agit repos
NAME        PATH                          BRANCH  WORKTREES  TASKS
openclaw    /home/user/projects/openclaw   main    2 active   1 pending
my-api      /home/user/projects/my-api     main    0 active   0 pending
```

4.4 agit spawn <repo> [--task description] [--branch name] [--agent id]

Create an isolated worktree for an agent. This is the core command.

```
$ agit spawn openclaw --task "refactor auth module" --agent claude-1
```

```
Created worktree: /home/user/projects/openclaw/.worktrees/agit-a1b2c3d4
  Branch: agit/refactor-auth-module-a1b2c3
  Agent:  claude-1
  Task:   refactor auth module

Agent can work in: /home/user/projects/openclaw/.worktrees/agit-a1b2c3d4
```

Behavior:

1. Creates a new branch from the repo default branch (or --from-branch)
2. Creates a Git worktree at `<repo>/.worktrees/agit-<short-uuid>`
3. Records worktree, agent assignment, and task in the registry
4. Returns the worktree path for the agent to use

4.5 agit status [repo]

Show all active worktrees, agents, and any detected conflicts.

```
$ agit status openclaw
REPO: openclaw (main)

ACTIVE WORKTREES:
  agit-a1b2c3d4  branch:agit/refactor-auth  agent:claude-1  task:refactor auth module
  agit-e5f6g7h8  branch:agit/add-tests      agent:cursor-1  task:add unit tests

CONFLICTS:
  WARNING: src/auth/handler.go modified in both agit-a1b2c3d4 and agit-e5f6g7h8

TASKS:
  [pending]     fix rate limiter (unclaimed)
  [in_progress] refactor auth module (claude-1)
  [in_progress] add unit tests (cursor-1)
```

4.6 agit conflicts [repo]

Check for overlapping file changes across all active worktrees in a repository.

```
$ agit conflicts openclaw
Scanning 2 active worktrees...

CONFLICT: src/auth/handler.go
  Modified in: agit-a1b2c3d4 (claude-1: refactor auth module)
  Modified in: agit-e5f6g7h8 (cursor-1: add unit tests)

CONFLICT: src/auth/middleware.go
  Modified in: agit-a1b2c3d4 (claude-1: refactor auth module)
  Modified in: agit-e5f6g7h8 (cursor-1: add unit tests)

2 conflicts detected across 2 files.
```

Implementation: For each active worktree, run `git diff --name-only <default-branch>...<worktree-branch>` to get modified files. Cross-reference across worktrees to find overlaps. Update file_touches table.

4.7 agit merge <worktree-id> [--strategy ours|theirs|manual]

Merge a worktree branch back into the base branch.

Behavior:

1. Runs conflict check first (unless `--skip-conflict-check`)
2. If conflicts detected, warns and requires --force or --strategy
3. Performs git merge into the base branch
4. Marks worktree status as completed
5. Optionally cleans up the worktree (--cleanup)

4.8 agit cleanup [--all] [--stale]

Remove completed or stale worktrees and their branches.

```
$ agit cleanup --stale
Removing 1 stale worktree(s):
  agit-x9y0z1w2 (openclaw) - last active 26h ago
Cleaned up 1 worktree(s).
```

4.9 agit tasks <repo> [--create description] [--claim id] [--complete id]

Manage tasks for a repository.

```
$ agit tasks openclaw --create "fix rate limiter bug"
Created task: t-a1b2c3d4 - fix rate limiter bug

$ agit tasks openclaw
ID          STATUS       AGENT     DESCRIPTION
t-a1b2c3d4  pending      -         fix rate limiter bug
t-e5f6g7h8  in_progress  claude-1  refactor auth module
```

4.10 agit serve [--transport stdio|sse] [--port 3847]

Start the MCP server. This is how agents connect to agit.

```
$ agit serve
agit MCP server running (stdio)
Agents can connect via MCP configuration.
```

5. MCP Server Specification

The MCP server is what makes agents infrastructure-aware. It exposes agit operations as MCP tools and the registry as MCP resources. Any MCP-compatible agent (Claude Code, Cursor, OpenClaw) can connect and immediately discover available repositories and coordinate work.

5.1 MCP Tools

Tools are actions agents can invoke.

Tool Name	Parameters	Returns
`agit_list_repos`	none	Array of {name, path, remote_url, default_branch, active_worktrees_count,

		pending_tasks_count}
`agit_repo_status`	`repo: string`	Detailed repo state: branches, worktrees, tasks, last sync
`agit_spawn_worktree`	`repo: string, task?: string, branch?: string`	{worktree_id, path, branch} - agent can start working at path
`agit_check_conflicts`	`worktree_id: string`	Array of {file_path, conflicting_worktrees[]} or empty if clean
`agit_list_tasks`	`repo: string, status?: string`	Array of tasks with their status and assignment
`agit_claim_task`	`task_id: string, agent_id: string`	{success, worktree_path} - auto-spawns worktree if needed
`agit_complete_task`	`task_id: string, result?: string`	{success} - marks task done, optionally records result
`agit_merge_worktree`	`worktree_id: string, strategy?: string`	{success, conflicts?} - merges and optionally cleans up
`agit_register_agent`	`name: string, type: string`	{agent_id} - registers agent in the system
`agit_heartbeat`	`agent_id: string`	{ok} - updates last_seen timestamp

5.2 MCP Resources

Resources provide read-only context that agents can access.

URI	Description
`agit://repos`	List of all registered repositories with summary stats
`agit://repos/{name}`	Detailed state of a specific repo
`agit://repos/{name}/worktrees`	Active worktrees for a repo
`agit://repos/{name}/tasks`	Tasks for a repo
`agit://agents`	All registered agents and their status
`agit://conflicts`	Current conflicts across all repos

5.3 Agent Connection Flow

When an agent connects via MCP, the typical flow is:

1. Agent calls `agit_register_agent` to announce itself
2. Agent calls `agit_list_repos` to discover available infrastructure
3. Agent calls `agit_list_tasks` to see what work is available
4. Agent calls `agit_claim_task` or `agit_spawn_worktree` to start working

5. Periodically calls `agit_check_conflicts` while working
6. Calls `agit_complete_task` and `agit_merge_worktree` when done

This means an agent never starts a session blind. It always knows what repos exist and what work is available.

5.4 MCP Configuration

Users add agit to their agent MCP config:

```
// Claude Code: ~/.claude/mcp.json
{
  "mcpServers": {
    "agit": {
      "command": "agit",
      "args": ["serve"]
    }
  }
}
```

6. Conflict Detection Engine

6.1 Approach

Rather than continuous file system watching (complex, resource-intensive), agit uses an on-demand approach: conflicts are detected by comparing git diffs across active worktrees.

6.2 Algorithm

1. For each active worktree in the target repo:
   - Run `git diff --name-only <default-branch>..<worktree-branch>`
   - Collect the set of modified files per worktree
2. Build a file-to-worktrees map:
   - For each file, record which worktrees have modified it
3. Files appearing in 2+ worktrees are conflicts
4. Update the file_touches table in the registry
5. Return conflict report with file paths, worktree IDs, agent names, and task descriptions

6.3 Conflict Severity

Severity	Condition	Action
Warning	Same file modified in 2+ worktrees	Notify, suggest coordination
High	Same function/block modified (future: AST-level diff)	Block merge until resolved
Critical	Conflicting changes to config, schema, or migration files	Require manual resolution

MVP implements Warning level only. Higher severity levels require AST-level analysis (future work).

7. Project Structure

```
agit/
  cmd/                          # CLI commands (Cobra)
    root.go                     # Root command, global flags, version
    init.go                     # agit init
    add.go                      # agit add <path>
    repos.go                    # agit repos
    spawn.go                    # agit spawn
    status.go                   # agit status
    conflicts.go                # agit conflicts
    merge.go                    # agit merge
    cleanup.go                  # agit cleanup
    tasks.go                    # agit tasks
    serve.go                    # agit serve (MCP server)
  internal/
    registry/                   # SQLite registry management
      db.go                     # DB init, migrations, connection pool
      repos.go                  # Repo CRUD operations
      worktrees.go              # Worktree CRUD operations
      agents.go                 # Agent tracking
      tasks.go                  # Task management
      touches.go                # File touch tracking
    git/                        # Git operations wrapper
      worktree.go               # Create, remove, list worktrees
      branch.go                 # Branch create, delete, list
      diff.go                   # Diff operations for conflict detection
      merge.go                  # Merge operations
      repo.go                   # Repo discovery (remote, default branch)
    conflicts/                  # Conflict detection engine
      detector.go               # Cross-worktree conflict analysis
      report.go                 # Conflict report formatting
    mcp/                        # MCP server implementation
      server.go                 # Server setup and lifecycle
      tools.go                  # Tool definitions and handlers
      resources.go              # Resource definitions and handlers
    config/                     # Configuration management
      config.go                 # TOML config parsing and defaults
  main.go                       # Entry point
  go.mod
  go.sum
  Makefile                      # Build, test, lint targets
  README.md
```

8. Dependencies

Package	Purpose	Notes
`github.com/spf13/cobra`	CLI framework	Industry standard for Go CLIs
`github.com/go-git/go-git/v5`	Pure Go Git implementation	No external git binary dependency
`github.com/mattn/go-sqlite3`	SQLite driver (CGO)	Or modernc.org/sqlite for pure

		Go
`github.com/pelletier/go-toml/v2`	TOML config parsing	v2 for TOML 1.0 compliance
`github.com/mark3labs/mcp-go`	Go MCP SDK	Supports stdio and SSE transports
`github.com/google/uuid`	UUID generation	For IDs across all tables
`github.com/fatih/color`	Terminal colors	For CLI output formatting
`github.com/olekukonez/tablewriter`	Terminal tables	For agit repos, agit status output

8.1 Build Requirements

- **Go 1.22+** (for range-over-func and enhanced stdlib)
- **CGO enabled** if using mattn/go-sqlite3. Alternative: modernc.org/sqlite (pure Go, no CGO)
- **Git 2.20+** installed on the system (for worktree operations via go-git)

9. Implementation Plan

Each phase is designed to be implementable in a single agent session (Claude Code, Codex, etc.) given this spec document as context.

Phase 1: Foundation

Goal: Basic project structure, config, and registry.

Files: `main.go, cmd/root.go, internal/config/config.go, internal/registry/db.go`

Deliverable: agit init works, creates ~/.agit/ with config and empty database.

Test: `agit init` creates expected file structure.

Phase 2: Repo Management

Goal: Register and list repositories.

Files: `cmd/add.go, cmd/repos.go, internal/registry/repos.go, internal/git/repo.go`

Deliverable: `agit add <path>` and `agit repos` work end-to-end.

Test: Register a real Git repo, verify it appears in the list.

Phase 3: Worktree Management

Goal: Spawn, list, and clean up worktrees.

Files: `cmd/spawn.go, cmd/status.go, cmd/cleanup.go, internal/registry/worktrees.go, internal/git/worktree.go, internal/git/branch.go`

Deliverable: `agit spawn` creates isolated worktrees, `agit status` shows them, `agit cleanup` removes them.

Test: Spawn 2 worktrees, verify they exist as separate directories with correct branches.

Phase 4: Conflict Detection

Goal: Detect overlapping file changes across worktrees.

Files: `cmd/conflicts.go, internal/conflicts/detector.go, internal/conflicts/report.go, internal/git/diff.go, internal/registry/touches.go`

Deliverable: `agit conflicts` identifies files modified in multiple active worktrees.

Test: Spawn 2 worktrees, modify same file in both, verify conflict is detected.

Phase 5: Task Coordination

Goal: Create, claim, and complete tasks.

Files: `cmd/tasks.go, internal/registry/tasks.go, internal/registry/agents.go`

Deliverable: Full task lifecycle works via CLI.

Test: Create task, claim it, complete it, verify state transitions.

Phase 6: Merge

Goal: Merge worktree branches back to base.

Files: `cmd/merge.go, internal/git/merge.go`

Deliverable: `agit merge` with pre-merge conflict check.

Test: Merge a worktree with no conflicts. Attempt merge with conflicts, verify warning.

Phase 7: MCP Server

Goal: Expose all operations via MCP for agent consumption.

Files: `cmd/serve.go, internal/mcp/server.go, internal/mcp/tools.go, internal/mcp/resources.go`

Deliverable: MCP server starts, agents can connect and use all tools.

Test: Connect a Claude Code session via MCP, verify agit_list_repos returns registered repos.

10. Usage Scenarios

Scenario 1: Single Developer, Multiple Agents

A developer has Claude Code and Cursor both configured with agit MCP. They register their main project:

```
$ agit add ~/projects/my-app --name my-app
$ agit tasks my-app --create "refactor authentication"
$ agit tasks my-app --create "add integration tests"
$ agit tasks my-app --create "update API documentation"
```

Claude Code connects via MCP, sees 3 pending tasks, claims "refactor authentication." agit auto-spawns a worktree. Meanwhile, Cursor connects, sees 2 remaining tasks, claims "add integration tests" in a separate worktree. Both work in parallel without conflicts.

Scenario 2: Conflict Prevention

Both agents happen to modify the same config file. When Claude Code calls agit_check_conflicts(), it gets a warning that src/config/auth.go is also modified in Cursor's worktree. Claude Code adjusts its approach to avoid the overlapping file.

Scenario 3: OpenClaw Agent with Persistent Context

An OpenClaw agent starts a new session. Instead of the developer having to explain what repos are available, the agent calls agit_list_repos() and immediately knows:

- openclaw repo at ~/projects/openclaw with 2 active worktrees
- my-api repo at ~/projects/my-api with 1 pending task
- docs repo at ~/projects/docs with no active work

The agent has full infrastructure awareness from the first message.

11. Future Work (Post-MVP)

- **AST-level conflict detection:** Parse modified files to detect function-level overlaps, not just file-level.
- **GitHub/GitLab integration:** Auto-import issues as tasks, auto-create PRs from completed worktrees.
- **Agent performance tracking:** Track completion rates, conflict rates, and merge success per agent type.
- **Remote registry sync:** Share the registry across machines for team use.
- **Pre-merge simulation:** Dry-run merges to predict conflicts before they happen.
- **TUI dashboard:** Real-time terminal UI showing all agents, worktrees, and tasks (using bubbletea).
- **Container isolation:** Option to run each worktree in a Docker container for full sandboxing.