Operation Manual — rag-poc (MVP)

Scope
- This document describes the current user-facing UI for tasks/jobs, the operational workflow for running and monitoring agent runs, and step-by-step instructions for running the sentinel smoke test introduced in the repository.

Audience
- Developers operating a local dev environment and SREs testing behavior in CI or local clusters.

1. User Interface (current MVP)

- Tasks List (`GET /api/tasks`)
- Displays a list of persisted `Task` records (id, title, status).
- Typical actions: create new task (POST to `/api/tasks` via UI or API), view details, trigger run.

- Task Detail / Run UI
- Shows task metadata (title, description, status) and the timeline of `Activity` events.
- A `Run` button triggers a `POST /run-agents` request for the given task payload.
- When run is invoked, the backend attempts distributed duplicate protection. The UI should reflect `running`/`failed`/`done` states based on SSE events or polling `/api/tasks`.

- Jobs / Worker View
- Displays worker (local or Celery) status: queued tasks, currently running jobs, and recent failures.
- When Celery is enabled, the worker service executes tasks and publishes events to Redis (if configured) so the UI can subscribe to updates across containers.

- Real-time updates (SSE)
- The frontend can open a Server-Sent Events connection to `GET /events/tasks/{task_id}` to receive real-time JSON events. Events include: agent activity, status updates, and structured progress messages.

2. AI-Assisted Coding and the Evolution to Intelligent Frameworks

AI-assisted coding has transformed modern development environments by embedding intelligent tools directly into IDEs and platforms like GitHub. In IDEs such as Visual Studio Code and JetBrains, solutions like GitHub Copilot, Amazon CodeWhisperer, and Tabnine provide context-aware code completion, natural language-to-code generation, and automated documentation. These tools streamline workflows by suggesting entire code blocks, improving debugging, and enabling rapid refactoring. On GitHub, AI extends beyond coding assistance to include Copilot Chat for conversational help, automated pull request summaries, CLI support, and security vulnerability fixes through Copilot for Security. Collectively, these innovations accelerate development, enhance code quality, and integrate seamlessly with cloud-based environments, signaling a future where AI-driven collaboration and automation become standard in software engineering.

The Intelligent Framework Advantage: Beyond Single-Context AI

While traditional AI coding assistants excel at generating code from current context, this intelligent framework represents a significant architectural evolution by implementing **multi-source Retrieval-Augmented Generation (RAG)** that combines temporal, semantic, and analytical intelligence:

Temporal Intelligence via Git Integration
- Traditional assistants analyze only the *current state* of code; this framework understands *code evolution

over time*
- Answers critical questions standard AI cannot: "When was this bug introduced?", "Who changed the authentication logic?", "What was modified in the last merge?"
- Provides cryptographically verified commit history with author attribution, timestamps, and exact diffs—not interpretations

Semantic Intelligence via Qdrant Vector Database
- While IDE assistants search within open files, this framework performs **cross-codebase semantic search** across the entire repository
- Discovers conceptually similar code patterns even when variable names differ
- Enables natural language queries like "Find all error handling patterns" without knowing specific file locations
- Ranks results by relevance, surfacing the most applicable code examples

Change-Aware Analysis
- Standard AI tools suggest code based on patterns; this framework explains *why code changed* by analyzing commit messages and diffs
- Performs impact assessment: "What else was modified when the API changed?"
- Traces code lineage: "How did this function evolve from version 1.0 to 2.0?"
- Provides root cause analysis for incidents by examining the exact changes that introduced issues

Multi-Dimensional Context for LLM Reasoning
- GitHub Copilot receives context from your current file; this framework provides the LLM with three perspectives simultaneously:
1. **Git metadata**: When/who/why changes occurred (temporal context)
2. **Qdrant search results**: Related code across the codebase (semantic context)
3. **Current code state**: Files and functions being analyzed (static context)
- Results in 8x more answerable question types with dramatically richer analysis

Intelligent Source Selection
The framework implements a 3-tier fallback strategy that adapts to query types:
- **Tier 1 (Git Tools)**: When analyzing specific commits, branches, or time periods
- **Tier 2 (Qdrant RAG)**: When finding similar patterns or performing semantic searches
- **Tier 3 (Explicit Files)**: When user specifies exact files to analyze

Real-World Comparison

Traditional AI Assistant Query: "Explain this authentication code"
- **Response**: Describes the current implementation based on visible code
- **Limitation**: No awareness of *when* it was added, *why* it was designed this way, or *what changed recently*

Intelligent Framework Query: "What is the root cause of commit 37c2ed14?"
- **Git Analysis**: "Merged ci/dispatch-qdrant branch on Nov 26, 2025 by reginaldrhoe. Added E2E CI tests with mock OpenAI. Files changed: lock_smoke_test.yml (+70 lines), openai_mock.py (+41 lines)"
- **Qdrant Context**: "Found similar testing patterns in test_crewai_adapter.py using mock implementations"
- **LLM Synthesis**: "The root cause was the need for CI testing without external API dependencies. The mock service enables deterministic offline testing following established patterns in the codebase."

Architectural Advantages for Software Engineering

1. **Compliance & Audit**: Verifiable change history with author attribution and timestamps
2. **Onboarding**: New developers can ask "What changed in the auth module recently?" and get temporal

analysis
3. **Code Review Automation**: Analyzes diffs with context from related historical changes
4. **Refactoring Safety**: Identifies similar code patterns that need updating across the codebase
5. **Incident Response**: Pinpoints exactly when and why bugs were introduced with commit-level precision
6. **Knowledge Preservation**: Captures the "why" behind changes through commit messages, not just the "what"

This multi-source architecture transforms AI from a code completion tool into a **comprehensive engineering intelligence platform** that understands not just *how* code works, but *why* it exists, *when* it changed, and *who* contributed—context essential for professional software development at scale.

For detailed architectural analysis, see `docs/ARCHITECTURE_ANALYSIS.md`.

---

3. Operational Workflow

This section documents the common sequence an operator or dev follows to run a task and monitor progress.

1) Prepare environment
- Ensure required services are running via `docker compose`: `redis`, `qdrant`, `mcp`, (optionally `worker` and `prometheus`).
- Confirm `REDIS_URL` and other env values are set in `.env` (or compose service environment).

2) Create or identify a Task
- Use the UI or API to create a `Task` record. Example (PowerShell):

Invoke-RestMethod -Uri http://localhost:8001/api/tasks -Method Post -Body (@{ title='smoke'; description='smoke' } | ConvertTo-Json) -ContentType 'application/json'

3) Start a run
- From the UI click Run, or call the run endpoint:

Invoke-RestMethod -Uri http://localhost:8001/run-agents -Method Post -Body (@{ id=11; title='Lock test task' } | ConvertTo-Json) -ContentType 'application/json'

4) Observe progress
- Open SSE connection to `GET /events/tasks/{id}` in the UI to receive events, or poll `/api/tasks/{id}`.
- Prometheus scrapes `/metrics` for counters and can alert on abnormal rates (e.g., many 409s).

5) Troubleshooting duplicate-run behavior
- The system uses a layered duplicate-protection strategy (up-front Redis check, tokenized redis lock, DB-level atomic update). If you see duplicate runs, check:
- Redis connectivity and TTLs (ensure `REDIS_URL` points to the running redis service).
- Sentinel logs inside the `mcp` container under `/tmp` (see smoke test below).

3. Smoke Test: sentinel-based lock test

Purpose
- Verify cross-process duplicate protection works: when two concurrent `POST /run-agents` calls target the

same task id, one should acquire the lock and the other should be rejected.

Files created by `mcp` (inside container)
- `/tmp/run_agents_entered.log` — each handler entry (timestamp + id)
- `/tmp/run_agents_lock_acquired.log` — records of locks acquired (method, token_len)
- `/tmp/run_agents_lock_conflict.log` — records of conflicts/up-front detections

Automated smoke test (PowerShell)
- A reusable script is included at `scripts/run_lock_smoke.ps1`. From the repo root run:

Run for task id 11
.\scripts\run_lock_smoke.ps1 -TaskId 11

What the script does
- Sends two near-concurrent POSTs to `POST /run-agents` for the same task id (one background job and one foreground request). It then reads the sentinel files from the `mcp` container and validates that at least one `LOCK_ACQUIRED` and one `LOCK_CONFLICT` entry exist.

Interpreting results
- PASS: script exits 0 and shows at least one lock-acquired and one conflict entry. This means cross-process protection triggered as expected.
- FAIL: missing entries indicate either `mcp` didn't run the sentinel-enabled code, Redis was not reachable, or the up-front check/lock flow fell back to DB path unexpectedly. Inspect container logs and sentinel files to diagnose.

Manual sentinel inspection
- You can manually inspect sentinels via:

docker compose ps -q mcp | ForEach-Object { docker exec $_ sh -lc "echo '--- /tmp/run_agents_entered.log ---'; cat /tmp/run_agents_entered.log || true; echo '--- /tmp/run_agents_lock_acquired.log ---'; cat /tmp/run_agents_lock_acquired.log || true; echo '--- /tmp/run_agents_lock_conflict.log ---'; cat /tmp/run_agents_lock_conflict.log || true" }

4. Operational checks and tips
- If the second request returns 200 (both succeed) during tests:
- Ensure `TEST_HOLD_SECONDS` is set in the `mcp` service env (increase to 30) so the first run is long enough to conflict deterministically.
- Verify Redis is reachable from the `mcp` container and `task:{id}:lock` key semantics work (use `redis-cli` inside container to inspect keys).

- To speed up developer iterations, consider mounting source into the `mcp` container during local dev so you don't need to rebuild the image for every change.

5. Where to find logs and artifacts
- FastAPI / uvicorn logs: visible from `docker compose logs mcp`.
- Sentinel files: `/tmp/run_agents_entered.log`, `/tmp/run_agents_lock_acquired.log`, `/tmp/run_agents_lock_conflict.log` inside the `mcp` container.
- Application metrics: `http://localhost:8001/metrics` (Prometheus format).

6. Next operational improvements (short list)
- Add a small UI panel for lock/conflict history for a selected task id.

- Convert sentinel smoke test to a Python script for CI portability and add a GitHub Actions job that runs it.
- Add a health check endpoint that validates Redis and DB connectivity and fails if either is unhealthy.

7. End-to-end and UI tests (new additions)

- **E2E integration script**: `scripts/run_e2e_integration.py` — a scaffold that:
- Creates a Task via `POST /api/tasks`.
- Ingests a small document via `POST /api/ingest`.
- Waits briefly for vector indexing, then calls the agent via `POST /run-agents` and asserts that retrieval text is visible in the response.
- Supports a deterministic mock mode so CI can run without an OpenAI key:
- Run with `--mock` or omit `OPENAI_API_KEY` in the environment to run in mock mode.
- Example (mock mode):

```
python scripts/run_e2e_integration.py --mock
```

- Example (live mode, requires app to call OpenAI internally):

```
ensure OPENAI_API_KEY is set in environment for the app
python scripts/run_e2e_integration.py
```

- **Playwright UI test scaffold**: `tests/ui/test_ui_agent_flow.py` (Python Playwright)
- Minimal scaffold that opens the frontend, fills a query input, and asserts that a visible response contains the expected retrieval text.
- Setup and run:

```
python -m pip install playwright pytest
playwright install
pytest tests/ui/test_ui_agent_flow.py -q
```

- Notes:
- Update the selectors in `tests/ui/test_ui_agent_flow.py` to match your UI (defaults assume `#agent-query`, `#agent-submit`, `#agent-response`).
- The UI test is optional for CI (it requires the frontend to be available at `http://localhost:3000` by default).

8. CI considerations

- The lock smoke-test workflow was updated to dynamically create a test Task at runtime and pass its id to the smoke-test. This makes CI runs reliable without relying on a hard-coded task id.
- If you want CI to run the E2E integration script:
- Either provide an `OPENAI_API_KEY` secret for real agent calls, or run the script with `--mock` to avoid external API calls.
- Example workflow step to run the E2E scaffold in mock mode:

```
- name: Run E2E scaffold (mock)
run: |
python3 scripts/run_e2e_integration.py --mock
```

- **In-app / local OpenAI mock server**
- A lightweight OpenAI-compatible mock server is provided at `mcp/openai_mock.py` and can be started on

port `1573` using the helper script:

python scripts/run_openai_mock.py

- Endpoints exposed (compatible shape):
- `POST http://localhost:1573/v1/chat/completions`
- `POST http://localhost:1573/v1/completions`
- `POST http://localhost:1573/v1/embeddings`

- Behavior:
- Deterministic replies: queries mentioning "sky" will return a reply containing "blue".
- Embeddings are deterministic vectors derived from the input text.

- Use this when running the frontend or CI to avoid real OpenAI calls. The frontend can be configured to point its OpenAI base URL to `http://localhost:1573`.

Contact and ownership
- File: `docs/OPERATION_MANUAL.md` — edit to extend UI details or to add organization-specific runbooks.

Maintenance & Recovery: Ingestion Control

When Git and Qdrant can drift (e.g., deleted files remain in Qdrant, or the `IndexedCommit` table is missing a record), operators can force synchronization using the secured admin endpoint.

Admin Endpoint
- `POST /admin/ingest` (requires `Authorization` token with `editor` role)
- Body:
- `repo_url` (required)
- `branch` (optional)
- `commit` (optional)
- `collection` (optional)
- `previous_commit` (optional)

Behavior
- If `IndexedCommit` is missing, the system performs a full index and writes the current commit to the database.
- If present, the system runs a git diff between previous and current, deletes points for removed files, and re-indexes only changed files.
- On diff failure, it falls back to full index.

Usage (PowerShell)
Full ingest
```
Invoke-RestMethod -Uri 'http://localhost:8001/admin/ingest' `
-Method POST `
-Headers @{ Authorization = 'Bearer ' } `
-ContentType 'application/json' `
-Body (@{ repo_url = 'https://github.com/owner/repo'; branch = 'main' } | ConvertTo-Json)
```

Incremental ingest
```
Invoke-RestMethod -Uri 'http://localhost:8001/admin/ingest' `
-Method POST `
-Headers @{ Authorization = 'Bearer ' } `
-ContentType 'application/json' `
```

-Body (@{ repo_url = 'https://github.com/owner/repo'; branch = 'main'; commit = ''; previous_commit = '' } | ConvertTo-Json)

Monitoring
- Check logs for `Incremental update from to ` and `Deleted points for: `.
- Verify `IndexedCommit` records and Qdrant collection counts.