RepDoctor2

AI-Powered Repository Management for Non-Developers

Product Specification v4.0 | February 2026

RepDoctor2 is a standalone tool for managing GitHub repository health. It scans all repositories under a GitHub account, uses AI to analyze what each branch contains and whether it should be integrated, and generates exact Claude Code commands for safe consolidation. It also teaches users how to configure their development workflow to prevent branch sprawl from recurring.

> Key architectural principle: RepDoctor2 is the brain, Claude Code is the hands. The app analyzes, recommends, and generates copy/paste-ready instructions. The user executes those instructions in Claude Code, where merge conflicts can be resolved, tests can be run, and rollbacks are easy. RepDoctor2 does not perform merges or branch deletions directly.

Design: 1980s retro terminal aesthetic — IBM Plex Mono, green phosphor on near-black. World-class usability with a retro coat of paint. This PDF uses a clean readable style; the app itself uses the retro terminal design.

Contents

1. Product Name & Problem Statement
2. User Story
3. Core Functionality & Architecture
4. AI Analysis Engine
5. The Setup Guide (Top Nav)
6. Archive & Reinstate System
7. Inputs & Outputs
8. Business Rules & Logic
9. Data Requirements
10. Design Direction & UI Specification
11. Integrations & Dependencies
12. Security Model

13. Out of Scope
14. Open Design Questions
15. Success Criteria
16. Deployment Notes
17. How to Run

1. Product Name & Problem Statement

Product: RepDoctor2 — AI-Powered Repository Management Tool

Problem: AI-assisted development tools like Claude Code create branches across sessions. Over time, repositories accumulate orphaned and divergent branches, and it becomes unclear which branch has the most current working code. Across a dozen or more repositories, this is unmanageable.

The deeper problem: even if a user can see all their branches, they don't know what's actually *in* each branch. Was it a half-finished experiment? A critical feature? A bug fix that never got merged? Without reading every commit and understanding the product context, cleanup is a guessing game.

And the *root cause* problem: there's no clear, accessible guidance inside these tools about how to prevent branch sprawl. Users don't know what to put in CLAUDE.md, how to configure Claude Code sessions to stay on main, or how to set up Claude Chat projects to reinforce good habits.

RepDoctor2 addresses all three layers: it diagnoses the current state across all repos, uses AI to understand what each branch contains and whether it matters, generates exact instructions for safe cleanup, and teaches the user how to prevent it from happening again.

2. User Story

Who: A product builder who uses Claude Code across multiple GitHub repositories. Not a professional developer. Understands product and business requirements deeply but relies on AI tools for implementation. Has accumulated branch sprawl across many repos and needs to regain control without deep Git expertise.

What they need to do:

- See every branch across all repos in one place
- Understand what each branch actually contains — in plain English, not commit hashes
- Know whether a branch has features that should be in main
- Get exact commands they can paste into Claude Code to consolidate safely
- Archive old branches so they're retrievable later if needed
- Configure their tools so future sessions always stay on main

The workflow: Open RepDoctor2 → see the full picture → read AI summaries → click to copy Claude Code instructions → paste into terminal → come back and verify everything is clean → configure tools for the future → done.

3. Core Functionality & Architecture

Architectural Principle: Analyze Here, Execute There

RepDoctor2 is a read-and-recommend tool. It connects to GitHub to gather data, uses the Anthropic API to analyze that data, and presents recommendations with copy/paste Claude Code commands. It does NOT write to GitHub directly except for creating archive tags (a safe, additive operation). All merges, branch deletions, and code changes happen in Claude Code where the user has full visibility and control.

> This matters because merging is where things go wrong. Claude Code can resolve conflicts, run tests, and roll back. A Flask web app cannot. Right tool for each job.

Phase 1: Scan & Audit

- Connect to GitHub via Personal Access Token (PAT)
- Fetch ALL repositories — public and private — owned by or accessible to the authenticated user
- For each repo, fetch all branches with metadata:
  - Commits ahead of default branch
  - Commits behind default branch
  - Last commit date and author
  - Whether it has an open pull request
  - Whether it has merge conflicts with the default branch
- Display results in a dashboard grid — one card per repo, drill into any for branch detail

Phase 2: AI Analysis

For each non-default branch, RepDoctor2 sends context to Claude via the Anthropic API and receives a structured analysis. This is the core differentiator — details in Section 4.

Phase 3: Triage & Classify

Each branch gets both an automatic classification AND an AI-generated summary:

- **SAFE TO DELETE** — Fully merged into the default branch. Zero unique commits.
- **AHEAD ONLY** — Has unique commits. Clean merge possible (fast-forward).
- **DIVERGED** — Both branch and default have unique commits. May conflict.
- **STALE** — No updates in 30+ days. Not merged.
- **ACTIVE PR** — Has an open pull request.

Phase 4: Generate Claude Code Instructions

For each branch, RepDoctor2 generates specific, copy/paste-ready Claude Code commands tailored to the branch's situation. These are not generic — they reference the actual repo name, branch name, local file path, and the AI's understanding of what the branch contains.

Example — AHEAD ONLY branch:

```
# RepDoctor2 instructions for: ministryfair / feature-signup
# AI Summary: Adds a volunteer signup form with email validation
# and a confirmation page. 4 commits ahead of main, no conflicts.
#
# Step 1: Open Claude Code in the project
cd ~/claudesync2/ministryfair && claude --continue

# Step 2: Paste this into Claude Code:
I need to merge the feature-signup branch into main. This branch
adds a volunteer signup form with email validation and a
confirmation page (4 commits ahead, no conflicts). Please:
1. Confirm we are on main (git checkout main && git pull)
2. Merge feature-signup into main (git merge feature-signup)
3. Run the app to verify nothing is broken
4. If everything works, push (git push origin main)
5. Delete the branch locally and remotely
6. Confirm the merge is complete
```

Example — DIVERGED branch:

```
# RepDoctor2 instructions for: parentpoint / dark-mode
# AI Summary: Implements dark mode across 6 components. Main has
# since received bug fixes to 2 of those same components.
# 7 ahead, 3 behind. Potential conflicts in Header.jsx, Footer.jsx
#
# Step 1: Open Claude Code in the project
cd ~/claudesync2/parentpoint && claude --continue

# Step 2: Paste this into Claude Code:
I need to merge the dark-mode branch into main. They have diverged.
dark-mode has theming across 6 components (7 commits ahead). Main
has bug fixes to Header.jsx and Footer.jsx (3 commits behind).
Conflicts likely in those two files. Please:
1. Checkout main and pull latest
2. Create a safety branch (git checkout -b merge-dark-mode-backup)
3. Switch back to main
4. Attempt the merge (git merge dark-mode)
5. Resolve any conflicts — keep dark mode changes but preserve
   the bug fixes from main
6. Test the app thoroughly
7. If good, push and clean up. If not, abort (git merge --abort)
```

Phase 5: Configure & Prevent

After cleanup, the Setup Guide (Section 5) walks the user through configuring CLAUDE.md, Claude Chat projects, and daily habits so branch sprawl doesn't recur.

4. AI Analysis Engine

The core intelligence of RepDoctor2. Uses the Anthropic API (Claude) to analyze branch content and generate actionable, personalized recommendations.

What Gets Sent to Claude

For each branch being analyzed, RepDoctor2 assembles a context package:

- **Commit history:** Commits on the branch not in the default branch — messages, authors, dates (via GitHub compare endpoint)
- **File diff summary:** Files changed with addition/deletion counts (not full file contents)
- **Product spec (optional):** If the user has uploaded a product spec for this repo, it's included so Claude can map changes to planned features
- **Branch metadata:** Name, repo, commits ahead/behind, last update, open PRs
- **Default branch recent history:** Last 10 commits on main so Claude understands what happened since the branch diverged
- **Previous scan results (if any):** So Claude can note trends (e.g., "this branch has grown from 3 to 7 commits since last scan")

What Claude Returns

A structured JSON response with:

- **plain_english_summary:** 2-3 sentences describing what this branch does, written for a non-developer
- **feature_assessment:** Should this be in main? SHOULD_MERGE | OPTIONAL | OBSOLETE | UNCLEAR
- **risk_level:** LOW (clean merge likely) | MEDIUM (some conflicts possible) | HIGH (significant divergence)
- **conflict_prediction:** Which files likely conflict and why
- **merge_strategy:** Recommended approach — fast-forward, merge, or rebase
- **claude_code_instructions:** Complete, copy/paste-ready text for Claude Code — customized to this branch, including cd, branch checks, merge steps, test reminders, cleanup, and abort commands
- **spec_alignment:** If a product spec was provided, which features this branch implements or relates to

The Prompt

System prompt tells Claude it is a Git branch analyst helping a non-developer:

- Write all summaries in plain English — no jargon, no commit hashes in user-facing text
- Explain what the branch DOES, not just what files it touches

- Make Claude Code instructions complete and copy/paste ready — include cd, branch verification, and rollback steps
- Map changes to spec features when a spec is available
- Be conservative with risk — when in doubt, flag MEDIUM or HIGH
- Return valid JSON only

Product Spec Integration

Users can upload or paste a product specification for each repo. This lets the AI understand not just what changed, but whether those changes align with the product direction.

- Specs stored locally in `data/specs/{repo-name}.md` (or .txt, .pdf)
- Upload/paste via Settings → Product Specs
- Included in AI prompt when analyzing that repo's branches
- If no spec provided, AI generates all other fields — spec alignment is omitted

Cost & Rate Management

- AI analysis is on-demand — user clicks [Analyze] per branch or [Analyze All] per repo
- Estimated token cost displayed before user confirms
- Results cached locally — re-analysis only needed if branch has new commits
- Running session cost total shown in footer
- Default model: Claude Sonnet (fast, cost-effective). Optional switch to Opus in Settings for complex repos.

IMPORTANT: The Anthropic API key is NEVER sent to GitHub. The GitHub PAT is NEVER sent to Anthropic. Each key is used only for its intended API.

5. The Setup Guide (Top Nav)

A permanent top-level navigation item — not buried in settings or help. Labeled "Setup Guide" in the top nav, always accessible. Contains everything a user needs to configure Claude Code, Claude Chat, and daily workflow habits to prevent branch sprawl. A user who has never touched CLAUDE.md can be fully set up in under 10 minutes.

5A. CLAUDE.md Configuration (for Claude Code)

A card with the header "Add this to your CLAUDE.md file". Contains a styled code block with a [Copy to Clipboard] button and step-by-step instructions:

- Step 1: Open Terminal
- Step 2: Navigate to your project folder (cd ~/your-project)
- Step 3: Open or create CLAUDE.md (nano CLAUDE.md or any editor)
- Step 4: Paste the copied text at the top
- Step 5: Save and close
- Step 6: Repeat for each repo

Recommended CLAUDE.md content:

```
## Branch Management Rules
- ALWAYS work on the main branch unless explicitly told otherwise
- NEVER create a new branch without asking the user first
- If you need to test something risky, ask before branching
- When starting a session, confirm you are on main before changes
- If you find yourself on a non-main branch, alert the user

## Commit Practices
- Commit frequently with descriptive messages
- Always push to origin after committing
- Include what changed AND why in commit messages

## Session Start Checklist
- Run: git branch --show-current (confirm main)
- Run: git status (confirm clean working directory)
- Run: git pull origin main (get latest)
- Only then proceed with the task
```

Dynamic repo checklist:

Because RepDoctor2 knows all repos, the Setup Guide shows a checklist: each repo name, local path (if detected), and CLAUDE.md status — green check (exists) or amber warning (missing). Common locations scanned: ~/claudesync2/, ~/Documents/, ~/Projects/, ~/Desktop/. User can set a custom root folder in Settings.

> Implementation note: Detecting local paths requires repos to be cloned locally. For repos only on GitHub, show "Not cloned locally" with a note. The CLAUDE.md check uses the GitHub API (GET /repos/{owner}/{repo}/contents/CLAUDE.md) so it works even without local clones.

5B. Claude Chat Project Instructions

A card: "Add this to your Claude Chat project instructions". [Copy to Clipboard] button and step-by-step instructions for claude.ai project settings.

Recommended content:

```
When discussing code changes or implementation:
- Always assume work happens on the main branch
- Do not suggest creating feature branches unless the user asks
- When writing specs for Claude Code, include a reminder to
  verify the active branch is main before starting
- If a task seems risky enough to warrant a branch, say so
  explicitly and let the user decide
- When generating terminal commands, start with:
  cd ~/claudesync2/[project] && git checkout main && git pull
```

5C. Claude Code Session Best Practices

Plain-English tips displayed as cards:

- **Always use --continue** when returning to a project — stays on the same branch with full context
- **Start from your project folder** — cd into the specific directory before launching Claude Code
- **Check your branch first** — ask Claude Code "What branch am I on?"
- **One project per session** — close and reopen in another folder to switch
- **Name your sessions** — /rename early for easier --resume later

Personalized quick-start commands:

RepDoctor2 generates a one-liner for each repo using the actual name and detected path. Each has its own [Copy] button:

```
cd ~/claudesync2/ministryfair && git checkout main && git pull origin main && claude --continue
cd ~/claudesync2/parentpoint && git checkout main && git pull origin main && claude --continue
cd ~/claudesync2/audioscribe && git checkout main && git pull origin main && claude --continue
```

6. Archive & Reinstate System

Old branches shouldn't just vanish. They contain work history, experiments, and context that might be valuable later. RepDoctor2 provides a structured archive system that preserves branches as Git tags — discoverable, reinstateable, and accessible to Claude Code.

How Archiving Works

- **Creates a Git tag** at the branch HEAD: `archive/[branch-name]/[YYYY-MM-DD]`
- **Tag message includes:** original branch name, archive date, unique commit count, last author, AI summary (if analyzed), optional user note
- **Pushes the tag to GitHub** — preserved remotely
- **Generates Claude Code instructions** for deleting the branch (user executes in terminal)
- **Logs the action** locally with all metadata

Archiving (tag creation) is done directly by RepDoctor2 because it's a safe, additive operation. Branch deletion is still done via Claude Code instructions because deletion is destructive.

Why Tags

Tags are permanent pointers to a specific commit. They don't move, can't be accidentally modified, don't clutter the branch list, push/pull naturally with Git, and are visible on GitHub's tags page. The active branch list stays clean while every commit is preserved.

The Archive Browser (Top Nav)

Lists all archive tags across all repos:

- Original branch name and repo
- Archive date
- AI summary (if analyzed before archiving)
- User note
- [Reinstate] → generates Claude Code instructions to recreate the branch
- [View on GitHub] → opens tag on GitHub
- Search and filter by repo, date, keyword

Reinstate Instructions (generated)

```
# Reinstate archived branch: feature-signup
# Archived: 2026-02-13
# Summary: Volunteer signup form with email validation

cd ~/claudesync2/ministryfair && claude --continue

# Paste into Claude Code:
I need to reinstate an archived branch. Please:
1. Create branch from archive tag:
   git branch feature-signup archive/feature-signup/2026-02-13
2. Push to origin: git push origin feature-signup
3. Checkout: git checkout feature-signup
4. Show contents: git log --oneline main..feature-signup
```

Claude Code Access Without Reinstating

Claude Code can read archived files directly without reinstating: `git show archive/feature-signup/2026-02-13:path/to/file`. The Setup Guide includes a tip about this so users know old work is never truly gone.

7. Inputs & Outputs

Inputs

- **GitHub PAT** — `repo` scope. Entered once, encrypted locally.
- **Anthropic API Key** — for AI analysis. Entered once, encrypted locally.
- **User password** — encrypts/decrypts both keys. Chosen on first run.
- **Local repo root folder** — optional. For CLAUDE.md detection and personalized commands.
- **Product specs** — optional, per-repo. For AI context.
- **User decisions** — trigger analysis, choose actions

Outputs

- **Repository Dashboard** — grid of all repos with branch counts and health indicators
- **AI Branch Analysis** — plain-English summaries, risk assessments, spec alignment
- **Claude Code Instructions** — copy/paste-ready commands for every recommended action
- **Archive Browser** — searchable archive of preserved branches
- **Setup Guide** — CLAUDE.md, Chat instructions, best practices with copy buttons
- **Consolidation Summary** — report of actions taken, exportable

8. Business Rules & Logic

Classification Rules

Condition	Classification	Color
Fully merged (0 ahead)	SAFE TO DELETE	Green
Ahead only, clean merge possible	AHEAD ONLY	Cyan
Both ahead and behind	DIVERGED	Amber
Last commit > 30 days, not merged	STALE	Red
Has open pull request	ACTIVE PR	Cyan bright

Safety Rules

- **RepDoctor2 never merges or deletes branches.** It generates Claude Code instructions.
- **Only direct write operation:** creating archive tags (safe, additive).
- **Generated instructions always include:** branch verification, backup for risky merges, test reminders, abort commands.
- **API key isolation:** GitHub PAT never touches Anthropic. Anthropic key never touches GitHub.

- **Default branch detection:** Uses actual default (main, master, develop). Notes non-standard names.
- **AI analysis on-demand:** User triggers explicitly. Cost visible before confirming.

Sorting

- Dashboard: repos by most branches first (highest cleanup need)
- Branch list: DIVERGED → AHEAD ONLY → STALE → SAFE TO DELETE → ACTIVE PR
- User toggle: by name, last updated, classification, AI risk level

9. Data Requirements

GitHub API Endpoints

- GET /user/repos — all repositories (public + private)
- GET /repos/{owner}/{repo}/branches — branches per repo
- GET /repos/{owner}/{repo}/compare/{base}...{head} — branch comparison
- GET /repos/{owner}/{repo}/pulls — open PRs
- GET /repos/{owner}/{repo}/tags — archive tags
- GET /repos/{owner}/{repo}/contents/CLAUDE.md — CLAUDE.md existence check
- POST /repos/{owner}/{repo}/git/tags + POST .../git/refs — create archive tags

Anthropic API

- POST /v1/messages — branch analysis (Sonnet default, Opus optional)
- Input: system prompt + branch context (commits, file changes, spec, metadata)
- Output: structured JSON (summary, assessment, risk, instructions, alignment)

Local Storage

- `config/credentials.enc` — encrypted PAT + Anthropic key (Fernet + PBKDF2)
- `config/preferences.json` — sort, excluded repos, local root, model preference
- `data/scan_history.json` — timestamped scan results
- `data/analysis_cache.json` — cached AI analyses keyed by branch + commit SHA
- `data/action_log.json` — all actions (instructions generated, archives, verifications)
- `data/specs/{repo-name}.md` — uploaded product specs

NOT Stored or Transmitted

- No source code downloaded or stored — analysis via API metadata only
- No data sent to third parties beyond GitHub and Anthropic APIs
- Keys never logged, displayed after entry, or cross-transmitted

10. Design Direction & UI Specification

1980s retro terminal aesthetic. Green phosphor text on near-black. All monospace (IBM Plex Mono primary, Fira Code fallback). 2025+ usability standards. No ASCII art. No decorative clutter. Clean, functional, distinctive.

Top Navigation

Nav Item	Destination	Notes
Dashboard	Repository grid	Default landing after login
Setup Guide	Config walkthrough	CLAUDE.md, Chat instructions, best practices
Archive	Archived branches	Search, browse, reinstate
Action Log	History of all actions	Instructions generated, archives, verifications
Settings	Preferences &amp; credentials	PAT, API key, specs, model, excluded repos, local root

Key UI Patterns

Branch Card — analyzed:

Classification badge (color-coded) | AI summary in plain English | Risk indicator | Spec alignment note (if available) | Prominent [Copy Claude Code Instructions] button. Instructions expand in a panel below the card. Panel includes [Copy All] and a [Mark as Done] checkbox. Done branches show a green check and move to bottom of list.

Branch Card — unanalyzed:

Classification badge | Commits ahead/behind | Last commit info | [Analyze with AI] button with cost estimate shown beside it.

Setup Guide cards:

Each card: descriptive header, code/text block with [Copy to Clipboard], numbered steps below. Three cards: CLAUDE.md, Claude Chat Instructions, Best Practices. Repo checklist with CLAUDE.md status per repo.

Instruction Panel (expanded):

Full Claude Code instruction block in a monospace code area. Includes a header comment block with the AI summary and context so the user (and Claude Code) understand what they're doing and why. [Copy All] button. [Mark as Done] checkbox.

Color Palette

Token	Hex	Usage
--bg-deep	#080a08	Page background
--bg-panel	#0b0e0b	Panels, cards

--bg-card	#0e120e	Elevated surfaces
--green-100	#33ff33	Primary accent, headings, active states
--green-80	#22cc22	Secondary headings, hover
--amber-100	#ffaa00	Warnings (DIVERGED, MEDIUM risk)
--red-100	#ff4040	Errors (STALE, HIGH risk)
--cyan-100	#40ffd0	Info (AHEAD ONLY, LOW risk, ACTIVE PR)
--text-primary	#b8d8b8	Body text
--text-secondary	#6a8a6a	Supporting text, metadata
--border	#1a2a1a	Card and section borders

Responsive

- Desktop (1024px+): 3-column repo grid, expandable branch panels
- Tablet (768-1023px): 2-column grid, branch detail replaces grid on drill-in
- Mobile (below 768px): single column, full-width cards and buttons

Retro Touches (subtle only)

- Very faint scanlines (opacity 0.03)
- Blinking cursor on active inputs
- All monospace, no exceptions
- No ASCII art, no decorative borders

11. Integrations & Dependencies

Required:

- **GitHub REST API v3** — repo/branch data, CLAUDE.md detection, archive tag creation
- **Anthropic Messages API** — AI branch analysis (Sonnet default, Opus optional)
- **Python 3.10+**
- **Flask** — local web server
- **Requests** — HTTP client for GitHub API
- **Anthropic Python SDK** — cleaner API integration
- **Cryptography (Fernet)** — credential encryption
- **IBM Plex Mono / Fira Code** — Google Fonts CDN

12. Security Model

- **Two keys, one password:** GitHub PAT and Anthropic key encrypted with user's password via Fernet + PBKDF2. Single encrypted file: config/credentials.enc.
- **Key isolation:** PAT only for GitHub calls. Anthropic key only for Anthropic calls. Never cross-transmitted.
- **In-memory only:** Decrypted keys held in memory during session. Cleared on exit.
- **No display after entry:** Keys masked. Never in logs, exports, or cross-API prompts.
- **Scope check:** Verify PAT has repo scope on auth. Warn on unnecessary scopes.
- **Recovery:** Forget password → delete credentials.enc → re-enter both keys. 5 minutes.
- **What goes to Anthropic:** Commit messages, file change summaries (names + line counts), branch metadata, product specs. Never source code contents. Never credentials.

13. Out of Scope (v1)

- Direct merge execution — all merges happen in Claude Code
- Direct branch deletion — generated as Claude Code instructions
- Code-level conflict resolution — Claude Code handles this
- CI/CD integration — no webhooks or automated triggers
- Multi-user / team features — single-user tool
- GitHub Organizations — personal repos only in v1
- Repo creation or deletion — branch management only
- Automated scheduling — scan is on-demand
- Full source code analysis — AI sees commits and file summaries, not file contents
- Claude Project / conversation integration — may be added in a future version

14. Open Design Questions

- **Batch instruction generation:** Generate a single mega-script for all branches in a repo? Powerful but could overwhelm. Maybe a "quick cleanup" mode for SAFE TO DELETE only.
- **Source code in AI context:** Currently AI sees commit messages and file change summaries. Option to include actual diffs? Better analysis but higher cost and tokens.
- **Verification loop:** After user executes instructions, a [Verify] button that re-scans just that repo to confirm success? Closes the loop nicely.
- **CLAUDE.md auto-setup:** Should the Setup Guide generate Claude Code instructions that create/update CLAUDE.md directly? Makes it executable, not just informational.
- **Instruction format:** Single copy-paste block vs. numbered steps? Single is faster; steps give more control.
- **Repo Doctor v1 migration:** If RepDoctor2 replaces Repo Doctor, should it import any existing Repo Doctor data (scan history, credential files)? Or clean start?
- **Future: Claude Project integration:** Could RepDoctor2 eventually ingest Claude project exports to map conversations to branches, similar to what Repo Doctor v1 planned?

15. Success Criteria

The tool is working when:

- User authenticates and sees all repos (public + private) in under 10 seconds
- Every branch is visible, classified, and has a plain-English recommendation
- AI analysis produces accurate, actionable summaries of branch contents
- Generated Claude Code instructions are complete, correct, and copy/paste ready
- User can go from "branch chaos" to "everything on main" in one session

- Archived branches are discoverable and can be reinstated
- Setup Guide gets CLAUDE.md configured across all repos in under 10 minutes
- No data lost — no working code accidentally deleted or overwritten
- API costs transparent — estimated and actual costs visible
- Works on Mac (primary) and PC (secondary)
- Non-developer can complete full cleanup without deep Git knowledge

16. Deployment Notes

- RepDoctor2 is a standalone project in its own GitHub repository: `repodoctor2`
- Not a branch or subdirectory of another project — it is the product
- Self-contained: own requirements.txt, own config directory, own data directory
- Runs locally on localhost:5001
- The security module (Fernet + PBKDF2 encryption) is built into the project — no external dependency on other tools

17. How to Run

First time:

- 1. Open Terminal
- 2. `cd ~/claudesync2`
- 3. `git clone https://github.com/[username]/repodoctor2.git`
- 4. `cd repodoctor2`
- 5. `pip install -r requirements.txt`
- 6. `python app.py`
- 7. Browser opens to localhost:5001
- 8. Enter GitHub PAT, Anthropic API key, choose a password

Every time after:

- 1. `cd ~/claudesync2/repodoctor2 && python app.py`
- 2. Enter password

This specification is self-contained. A developer or Claude Code session can build RepDoctor2 from this document without needing the original conversation. The retro terminal design system (color palette, typography, layout patterns) is fully specified in Section 10.