The GitHub Automations Playbook

Build your first self-running script in one afternoon using Claude Code, GitHub Actions, and three free tools

Charles Dove, C&C; Strategic Consulting

April 2026

What You Are Building

You have a task that you hate doing. Maybe you check a list of prices every morning. Maybe you copy data between two apps. Maybe you want new content to publish itself overnight while you sleep.

Whatever it is, the fix is the same shape: write a small script, tell GitHub to run it on a schedule, and never think about the task again. That is automation. It is the closest thing to a free employee you will ever get.

This playbook walks you through building your first automation from zero. You do not need to know how to code. You do not need to set up a server. You do not need to pay for hosting. You need Claude Code, a GitHub repo, and about an hour of focus.

How it works, the one-minute version

- Pick something you do by hand that could be code
- Open Claude Code inside your GitHub repo
- Tell it what you want in plain English
- Claude writes the script and the workflow file
- Add any secrets with one command
- Push to main and watch the first run
- Tweak until it is right, then forget it exists

That is the whole playbook in seven bullets. The rest of this document is details. By the end you will have one automation running live on GitHub's servers, and the pattern you learned will apply to the next fifty automations you build.

What You Need Before Starting

Three tools and one idea. Get them ready before you start. Missing any of these and the steps below will not work.

1. A GitHub account with at least one repo

A GitHub repo is where your code lives. It is also where GitHub Actions (the free scheduler and compute service we are using) actually runs. You need both pieces. If you have never used GitHub: go to github.com, make a free account, click New Repository, give it a name like my-automations, mark it Private, and you are done. The repo starts empty. That is fine.

2. GitHub CLI (gh) installed and authenticated

The GitHub CLI is a terminal tool called gh. It lets you talk to GitHub from your computer without opening a browser for every action. You will use it to set secrets, trigger workflow runs, and read logs. Install it with brew install gh on a Mac, or download it from cli.github.com on Windows. Then run gh auth login and follow the prompts. When you are done, gh auth status should show you logged in as your GitHub username.

3. Claude Code installed

Claude Code is the AI tool that writes the scripts for you. Get it from claude.com/code. It runs in your terminal alongside gh. You log in with the same Anthropic account you use for claude.ai. Claude Code reads your existing files and matches your style, so the more of your existing project it can see, the better.

4. A real pain you want to stop doing by hand

This is the one most people skip, and it is the one that actually matters. Do not start with "I want to try automation." Start with "I do X every day and it takes Y minutes and I am tired of it." The more specific you are about the pain, the better the automation will be. Vague input gives you vague output.

Why GitHub Actions (Not Your Mac, Not a Server)

There are three common places to run an automation. GitHub Actions wins for most jobs. Here is why, and when you might pick a different option.

Option A: Your own computer

You could write the script and run it on your Mac via cron. This works, but only when your Mac is awake, only when it has internet, and only if you remember to re-enable it after a reboot. Your automation is tied to one physical machine. If your Mac dies, the automation dies. Good for: personal stuff with private data you do not want leaving your machine.

Option B: A VPS (Virtual Private Server)

You could rent a Linux server from Hetzner or DigitalOcean for $5 a month and run your scripts there. This works great, but now you are a systems administrator. SSH keys, security updates, firewall rules, uptime monitoring. That is a whole second job you probably do not want. Good for: 24/7 services, webhooks that need to respond instantly, team-wide tools.

Option C: GitHub Actions

Free for up to 2,000 minutes per month on private repos, way more than most automations need. Your code lives in the same place as the script, so version history is baked in. Logs and run history are free. You can trigger runs manually from the website or the gh CLI. You can run on cron, on every push, or on webhook.

Good for: almost everything else. Daily jobs, weekly reports, triggered cleanups, API syncs, content publishing, automated tests, report generation. This playbook assumes you pick Option C. Start there. You can always move to a VPS later if you outgrow the free tier.

Step 1: Pick a Problem Worth Automating

The best automations solve a real pain. Vague automations solve vague problems and nobody uses them. Here is how to find the good ones.

Ask yourself these three questions

- What do I do every week that takes more than 15 minutes and does not need my judgment?
- What task do I put off because it is boring, not because it is hard?
- What would I pay a virtual assistant $20 to do if I could find one?

The honest answers point you at good candidates. A job that needs your creativity and judgment is NOT a good automation target, at least not yet. A job you would rather not do but anyone could do with the right recipe is a perfect automation target.

Ten automation ideas to spark thinking

- Auto-publish YouTube videos as blog posts on your site
- Scrape competitor prices every morning and email you what changed
- Check your website for broken links every Sunday night
- Pull new form submissions and drop them into a Google Sheet
- Send yourself a daily summary of new comments across platforms
- Auto-tag GitHub issues older than 30 days as stale
- Sync contacts between two different CRM systems
- Generate a weekly revenue report from your Stripe dashboard
- Back up files from Google Drive to a GitHub repo each week
- Auto-format and commit code changes nightly

Pick ONE idea. Not three, not five. One. You can always add more automations later. The goal this afternoon is to ship one working automation, not to plan twenty that never get built.

Step 2: Open Claude Code Inside Your Repo

This part matters: you open Claude Code inside the repo folder, not a blank folder somewhere else. Claude Code reads your existing files to match your style. If your repo already has Python scripts, a folder structure, or a README with your preferences, Claude picks up on all of that automatically.

Clone your repo to your computer, cd into it, and run claude. That is all. Claude Code opens in that repo's folder and can now see every file.

The prompt template that works

Paste this into Claude Code, filling in the parts in brackets:

```
I want to build a GitHub Action that does [WHAT].

Trigger: [cron schedule / on push / manual only]
Inputs needed: [what data it needs to work with]
Secrets required: [API keys, tokens, etc.]
Output: [where results go: file, email, another API]

The script should:
- Live in scripts/[name]/
- Use Python with minimal dependencies
- Have a --dry-run flag for testing
- Log clearly so I can debug via gh run view --log
- Fail loudly: print stderr, not just exit 1

The workflow at .github/workflows/[name].yml should:
- Run on the trigger above
- Have a manual workflow_dispatch trigger so I can test
- Use GITHUB_TOKEN for any git operations
```

Claude will ask clarifying questions. Answer them honestly. If you are not sure about something, pick the simpler option.

Step 3: Let Claude Scaffold the Files

After your prompt, Claude Code builds the project. For a typical small automation it creates a file tree that looks like this:

```
scripts/[your-name]/
main.py # the orchestrator
[helper1].py # one module per responsibility
[helper2].py
requirements.txt # dependencies, pinned
README.md # how to run locally
.env.example # template for local secrets

.github/workflows/
[your-name].yml # the Action definition
```

Red flags to stop and question

Do NOT accept the first draft blindly. Walk through the files Claude created and watch for these five red flags:

- No error handling at all: real code has retries and fallbacks. If yours has none, tell Claude to add them.
- Bare except: blocks that swallow errors: tell Claude to print stderr and name the exception types.
- Secrets hardcoded in the script: never. Ask Claude to read them from environment variables instead.
- No --dry-run mode: you need a way to test without doing real work. Add one.
- No workflow_dispatch trigger in the YAML: you need a manual button for testing. Add one.

All of these are fixable by telling Claude to fix them. You are the senior engineer on this project and Claude is your junior. Review the work. Send it back for revisions. Do not accept sloppy first drafts just because they compile.

Step 4: Secrets Go in gh CLI, Not in Code

Never put secrets in code. Never. If your script needs an API key, a token, a password, or anything else sensitive, it reads the value from an environment variable. The environment variable is populated from GitHub's Secrets vault at runtime. You control the vault via the gh CLI.

The two-command setup

First, create a .env file locally for testing (add .env to your .gitignore before you save anything into it, so it does not get accidentally committed):

```
ANTHROPIC_API_KEY=sk-ant-your-key-here
OTHER_SECRET=whatever-other-value
```

Then push all of it to GitHub's secret store with one command:

```
gh secret set --env-file scripts/[name]/.env \
-R [username]/[repo-name]
```

Done. Every secret in the file is now available to your Action at runtime as an environment variable. Reference them in your workflow YAML under the env: block, and Python reads them with os.environ.get("ANTHROPIC_API_KEY").

What GITHUB_TOKEN is and why you do not create it

GitHub auto-provides a special secret called GITHUB_TOKEN inside every Action run. It has permission to push to your repo, comment on pull requests, and do most GitHub API calls. You do not create it, you do not set it. You just reference it in the workflow YAML as ${{ secrets.GITHUB_TOKEN }}. Use it for anything that touches your own repo. Use your own secrets for anything that talks to outside services like Anthropic, Stripe, or third-party APIs.

Step 5: Test Before You Trust

The number one mistake people make with new automations is letting the cron run without ever testing manually. Do not do that. You will lose a day chasing a silent bug that shows up only on the runner and never on your laptop.

The first-test checklist

- Push the code to main first. The workflow file needs to exist on the default branch before it shows up in the Actions tab.
- Go to the Actions tab on GitHub. Click your workflow name. Click Run workflow. Set dry_run to true (if you added that input). Click the green button.
- Watch the logs. Each step shows green or red. Click into any step to read the output. Errors are usually in red at the bottom.
- Check that dry_run=true did NOT actually publish, email, commit, or do any side effect. Dry run proves the code path works without consequences.
- Now run it again with dry_run=false and watch carefully. This is the real thing.

Using gh to debug from your terminal

Clicking around the Actions tab gets old fast. The gh CLI gives you much quicker feedback from the terminal. Two commands cover 90 percent of what you need:

```
gh run list --workflow=[name].yml --limit 5
gh run view [run-id] --log
```

The first command shows the last five runs of your workflow with their status. The second command dumps the full log of any run as text, which is much faster to grep through than clicking around in the web UI when you are debugging.

The Four Things That Will Break It

Every automation breaks eventually. Here are the four most common causes and the exact fix for each one. Know these before they happen and you save yourself hours of debugging.

1. The green checkmark that lies (silent failures)

A workflow shows green but nothing actually happened. This usually means your script caught an error and exited cleanly instead of crashing. Fix: always print stderr from failed subprocess calls, log a clear "skipping X because Y" message every time you bail out, and treat missing outputs as a failure condition, not a quiet success.

2. Cloud IP blocks on third-party services

GitHub Actions runs on Microsoft Azure IPs. Services like YouTube, Instagram, and some news sites aggressively block Azure IPs to stop scrapers. Symptom: your script works on your Mac but fails in the Action with a 403, a captcha, or a "bot detected" error. Fix: use a residential proxy service like Webshare (free tier works for most low-volume jobs), or move the script to your home Mac and run it via launchd instead.

3. Secrets accidentally committed to the repo

You pasted a real API key into a config file, pushed to GitHub, and now it is public. This is recoverable but annoying. Rotate the key immediately at the provider (create a new one, delete the old one). Then remove it from your repo history with git filter-repo or BFG Repo Cleaner. Next time, use .env files and .gitignore.

4. Missing permissions on GITHUB_TOKEN

You want your Action to push commits but it fails with "permission denied." Fix: add permissions: contents: write to the job in your workflow YAML. Each permission you need is a one-line entry. Without this line, GITHUB_TOKEN is read-only by default and your Action cannot modify the repo it lives in.

What to Automate Next

Once your first automation is running, the same pattern handles a dozen other jobs. Same file structure, same gh secret set, same workflow_dispatch testing flow, same debugging commands. Pick another pain from your week and repeat.

Good next projects

- Content reposter: every YouTube video becomes a LinkedIn post, a Twitter thread, a blog post, and a carousel, all from the same transcript.
- Lead enricher: every new form submission gets auto-enriched with company size, LinkedIn profile, industry, and lands in your CRM.
- Price watcher: track prices on competitor sites, get an email if anything changes by more than 5 percent.
- Meeting note summarizer: transcripts from Zoom or Otter go in, summary, action items, and follow-up email drafts come out.
- Calendar auditor: every Monday morning, check your scheduled posts against your content rules and flag anything that does not fit.
- Inbox triage: label incoming emails by intent (sales, support, personal, spam) and route urgent ones to Slack.

Each one follows the same shape as the automation you just built. Swap the inputs, swap the API call, swap the output destination. The orchestrator, the secrets handling, the workflow YAML, the testing flow, and the debugging commands all stay almost identical.

The biggest mistake you can make right now

Do not try to build the perfect automation on your first try. Ship something that works 80 percent of the time on real data, watch it run for a week, and fix the gaps you did not see coming. The goal this week is "a new process that runs without me," not "bulletproof production code that handles every edge case " I can imagine." Perfect is the enemy of shipped. Ship the rough version. Tune it based on reality, not on what you imagined might go wrong.

Ready to Go Deeper?

You built one automation. The next fifty use the same pattern. Here are the two ways to get faster at this.

The Full Training and Skill Library

CC Strategic AI Premium gives you every Claude Code skill, every template, and every live build I run, applied to your business.

https://www.skool.com/cc-strategic-ai/plans

Work With Me 1-on-1

Prefer it built for you, or want a direct walkthrough tuned to your business? Book a 1-on-1 strategy call.

https://buy.stripe.com/5kQ9ATa7kfTPdeu1xj1Fe00