ChatGPT

Cline CLI Daily Workflow Recipes

Overview of the Cline CLI

The Cline CLI runs AI coding agents directly in your terminal and allows you to pipe Git diffs for automated code reviews, run multiple instances in parallel and integrate with existing shell workflows [1]. It supports model providers such as Anthropic, OpenAI, OpenRouter and many others; you authenticate via `cline auth` during installation [2]. The CLI is designed for automation and scriptability, supporting interactive sessions, headless one-shot tasks and multi-instance parallelization [3]. Common use cases include daily code maintenance (linting, vulnerability scans and dependency updates), multi-instance development and custom workflows that integrate with tools like GitHub CLI (`gh`), npm, Docker and Kubernetes [4].

Choosing the right flow

- **Interactive (Ask) Flow** – starts an interactive chat session where Cline proposes a plan you can review and approve before execution [5]. Use this for exploratory tasks or when you want oversight.
- **Do (headless) Flow** – run a single command that plans and executes autonomously using the `-y` (YOLO) flag [6]. Perfect for automation or CI pipelines, but it trades oversight for speed.
- **Compose (workflow) Flow** – define reusable multi-step processes in Markdown. Cline interprets these files and runs the steps to automate repetitive tasks [7].
- **Multi-Instance** – create separate Cline instances to run tasks in parallel (e.g., separate front-end and back-end tasks) [8].

Custom Daily Workflows

1. Daily Pull-Request Review Workflow

Purpose – Automate the process of reviewing GitHub pull requests by gathering details, examining changes and preparing review actions.

Prerequisites

- Cline CLI installed and authenticated with your preferred model provider.
- GitHub CLI (`gh`) installed and authenticated.
- A Git repository with open pull requests.

Steps

1. **Create Workflow File** – In your project's root, create `.clinerules/workflows/pr-review.md` [9].
2. **Gather PR Information** – Use `gh pr view PR_NUMBER --json title,body,files` to fetch the pull request title, description and list of changed files [10].

3. **Examine Modified Files** – Execute `gh pr diff PR_NUMBER` to view the unified diff of code changes [11].
4. **Analyze Changes** – Direct Cline to look for bugs (logic errors), performance issues and security vulnerabilities in the diff [12].
5. **Confirm Assessment** – Insert an `<ask_followup_question>` block that presents the analysis and asks how to proceed (approve, request changes, comment or do nothing) [13].
6. **Execute Review** – Based on the selected option, run the appropriate `gh pr review` command with an approval, request-changes or comment message [14].

Invoking this workflow via `/pr-review.md 42` triggers the entire sequence [15]. The workflow provides consistent reviews and reduces manual effort.

2. Daily Test Runner and Summary

Purpose – Run your entire test suite and summarize failures so that you can address issues quickly.

Prerequisites

- Cline CLI installed.
- Test command (e.g., `npm test` or `pytest`) available in the repository.
- Optionally, configure this workflow for headless execution with `cline task new -y` for unattended runs [6].

Steps

1. **Define the Task** – Create a workflow file (e.g., `.clinerules/workflows/run-tests.md`) with instructions to run your test command using an `<execute_command>` step.
2. **Run Tests** – Use `execute_command` to run `npm test` (or your test runner). In headless mode, the `-y` flag allows Cline to plan and execute autonomously [6].
3. **Analyze Results** – After execution, instruct Cline to parse the test output and identify failing test cases. Use bullet points to list each failed test and summarise the error messages.
4. **Summarise for Developer** – Add an `<ask_followup_question>` step prompting: “Tests finished – would you like me to open the failing files, suggest fixes, or just log the summary?”
5. **Optional Fix** – If you choose to fix, Cline can enter interactive mode to propose changes and run tests again until they pass.

You can schedule this workflow to run nightly via a cron job using the headless flow (`cline task new -y "Run tests and summarise failures"`), and watch progress with `cline task view --follow` [16].

3. Daily Linting and Security Scan

Purpose – Ensure code quality and security by running linters and vulnerability scanners across the codebase.

Prerequisites

- Cline CLI installed.
- Linting tools (e.g., ESLint for JavaScript, flake8 for Python) and vulnerability scanners (`npm audit`, `yarn audit`, `safety`, etc.) installed.

Steps

1. **Workflow Creation** – Save a file `.clinerules/workflows/code-quality.md`.
2. **Run Linters** – Use `<execute_command>` steps to run `npm run lint` or the relevant linter commands for each language; capture and summarise any warnings or errors.
3. **Run Security Scans** – Add steps to run `npm audit` (Node.js) or `pip list --outdated` / `safety` for Python; summarise vulnerabilities and outdated dependencies [17].
4. **Compile Report** – Instruct Cline to collect results into a markdown section. Use `write_to_file` to append the results to `reports/daily-code-quality.md` with the date.
5. **Ask for Next Step** – Add an `<ask_followup_question>` asking whether to automatically fix linting issues or update dependencies. If the user agrees, use `execute_command` to run `npm audit fix`, `eslint --fix`, etc., in interactive or YOLO mode depending on risk appetite.

This workflow helps maintain code health by automating quality checks. It leverages the CLI's ability to execute shell commands and summarise results [7].

4. Automated Daily Changelog Generation

Purpose – Produce a daily changelog of what you worked on by extracting recent commit messages and asking for a summary.

Prerequisites

- Git repository with commits.
- Cline CLI installed.

Steps

1. **Create** `daily-changelog.md` – Place this file in `.clinerules/workflows/` or your global workflows directory [18].
2. **Check Recent Commits** – Run a command such as `git log --author="$(git config user.name)" --since="yesterday" --oneline` to list commits from the previous day [19].
3. **Present Commits** – Instruct Cline to present the list of commit messages to you and ask for a brief summary of your work [20].
4. **Create or Append Changelog** – Append a section to `changelog.md` with the current date, list of commits and your summary [21].

This simple workflow reduces the cognitive load of preparing daily reports and ensures that the changelog is maintained consistently [22].

5. Version Bump & Release Preparation

Purpose – Automate version bumping, testing and releasing of a package to maintain consistent releases without forgetting critical steps.

Prerequisites

- Project uses a version file (`package.json`, `pyproject.toml`, etc.).
- Git is configured and you have permissions to push tags.

Steps

1. **Bump Version** – Use `read_file` to open the version file and modify the version number (for example, from `1.0.0` to `1.0.1`).
2. **Run Tests and Lint** – Execute your test suite and linters to ensure the new version does not break anything; summarise failures.
3. **Update Changelog** – Insert a new section in `CHANGELOG.md` describing the changes in this release. Use `write_to_file` to update the file.
4. **Commit & Tag** – Execute `git commit -am "v1.0.1"` and `git tag v1.0.1` [23]; then run `git push origin main --tags` to push both commit and tag [24].
5. **Confirm Publishing** – If your project is published to an artifact repository (e.g., npm, PyPI), add steps to run `npm publish` or `poetry publish`. Use `<ask_followup_question>` to confirm before publishing.

This workflow encapsulates release tasks into a repeatable process [25].

6. Feature Branch Setup

Purpose – Quickly create a new feature branch, scaffold the needed files and open a new Cline instance for parallel development.

Prerequisites

- Cline CLI installed.
- Git configured.

Steps

1. **Prompt for Branch Name** – Add an `<ask_followup_question>` to ask for the feature name.
2. **Create Branch** – Use `execute_command` to run `git checkout -b feature/<branch-name>`.
3. **Scaffold Files** – Instruct Cline to create boilerplate files (e.g., `src/components/<Feature>.jsx`, test files). This can be as simple as instructing Cline: “Generate a React component and its corresponding test.”
4. **Open New Instance** – Call `cline instance new` to create a parallel instance [26]; then run a headless task to build the feature (`cline task new -y "Implement the <feature> component"`). This isolates work from your main development context.
5. **Push Branch** – After scaffolding, run `git push -u origin feature/<branch-name>`.

This workflow standardizes branch creation and scaffolding across your team.

7. Documentation Generation

Purpose – Ensure that every function or module has up-to-date docstrings by generating documentation automatically.

Prerequisites

- Cline CLI installed.
- A codebase with functions or modules lacking documentation.

Steps

1. **Define Task** – Use headless mode: `cline task new -y "Add JSDoc comments to all functions in src/"` [27]. You can also embed this in a workflow file.
2. **Search for Undocumented Functions** – Let Cline search files for functions without documentation (e.g., use `search_files` for `function` keywords and check preceding comments).
3. **Insert Documentation** – Use `write_to_file` to insert or update docstrings based on the function signature and content. Cline's AI will infer parameters and return values.
4. **Report** – Summarise how many functions were documented and list any files that need manual attention.

This workflow uses the Do flow for autonomy and ensures consistent documentation across your codebase [27].

8. Parallel Front-End/Back-End Development

Purpose – Run front-end and back-end tasks simultaneously using multi-instance support.

Prerequisites

- Cline CLI installed.
- A monorepo or project with separate front-end and back-end directories.

Steps

1. **Create Front-End Instance** – Run `cline instance new` to create the first instance and note the returned address [26].
2. **Launch Front-End Task** – Attach a task to the first instance: `cline task new -y "Build React components"` [28].
3. **Create Back-End Instance** – Execute `cline instance new --default` to create a second instance and set it as the default [29].
4. **Launch Back-End Task** – Run `cline task new -y "Implement API endpoints"` on the default instance [30].
5. **Monitor Instances** – Use `cline instances list` to see all running instances [31]; use `cline instances kill -a` to stop them when done [32].

This workflow keeps contexts separate and accelerates development by allowing different parts of your stack to be worked on concurrently.

9. Dependency Update and Security Patch Workflow

Purpose – Automatically update outdated dependencies and apply available security patches.

Prerequisites

- Cline CLI installed.
- Package manager commands (`npm update`, `pip list --outdated`, `composer update`) available.

Steps

1. **Identify Outdated Packages** – Use `execute_command` to run the appropriate command for your language (e.g., `npm outdated` or `pip list --outdated`) and parse the output.
2. **Update Packages** – Instruct Cline to update packages one by one, ensuring compatibility. For Node.js, run `npm update <package>`.
3. **Run Tests** – After updates, run the test suite to verify nothing is broken [33].
4. **Commit Changes** – Use `git commit` with a message like "chore: update dependencies" and push the changes.
5. **Optional Security Scan** – Integrate with the security scan workflow to re-scan for vulnerabilities after updates.

Automating dependency updates helps maintain security and code quality and fits into daily maintenance tasks [17].

10. Daily Stand-Up Report Generator

Purpose – Gather your daily commits, summarize what you accomplished and produce a stand-up report or Slack message.

Prerequisites

- Git repository with commits.
- Slack MCP integration configured if you want automatic posting (optional).

Steps

1. **Fetch Commits** – Run `git log --author="$(git config user.name)" --since="09:00" --until="now" --oneline` to collect today's commits.
2. **Summarize Work** – Present the list of commits and ask the user for a short narrative of tasks completed.
3. **Compose Report** – Format a stand-up report including date, commit summaries, tasks in progress and blockers; write it to `standup.md` or send via Slack through an MCP tool.

4. **Ask for Confirmation** – Use `<ask_followup_question>` to confirm sending; if approved, use the Slack MCP to post the report to your team channel.

This workflow turns commit history into actionable daily updates, ensuring nothing gets missed in stand-ups.

Best Practices for Building Workflows

- **Leverage Cline for Workflow Creation** – Use Cline to help build workflows. The `create-new-workflow.md` file guides you through naming, describing objectives and listing steps; Cline then generates the workflow structure [34].
- **Store Workflows Properly** – Place project-specific workflows under `.clinerules/workflows/` and global workflows under `~/Documents/Cline/Workflows/` to make them available across projects [18].
- **Start Simple and Be Modular** – Begin with small workflows and combine them later. Break complex tasks into reusable pieces for easier maintenance [35].
- **Comment Your Steps** – Document why each step exists to aid future maintainers and to give Cline context for decisions [36].
- **Version Control** – Check workflow files into your Git repository so they are versioned and can be reviewed and updated with the codebase [37].
- **Be Specific with Tools** – When instructing Cline, specify which tool to use (e.g., "use `search_files` to find `UserController` in `src/controllers`") to ensure accurate execution [38].

Conclusion

The Cline CLI enables rich automation for daily software engineering activities. By understanding its core flows—interactive chat, headless single-shot execution and multi-instance parallelization—and by composing workflows in Markdown, you can automate pull-request reviews, testing, linting, documentation, releases and more. Leveraging Cline not only standardizes processes but also reduces cognitive load, enabling developers to focus on higher-value work [22]. These custom workflows serve as templates that you can tailor to your team's specific needs and integrate into existing CI/CD pipelines or daily routines.

---

[1] [2] [4] [17] [33] Overview - Cline
https://docs.cline.bot/cline-cli/overview

[3] [5] [6] [8] [16] [26] [27] [28] [29] [30] [31] [32] Three Core Flows - Cline
https://docs.cline.bot/cline-cli/three-core-flows

[7] [18] [19] [20] [21] [22] [23] [24] [25] Workflows Overview - Cline
https://docs.cline.bot/features/slash-commands/workflows

[9] [10] [11] [12] [13] [14] [15] Workflows Quick Start - Cline
https://docs.cline.bot/features/slash-commands/workflows/quickstart

[34] [35] [36] [37] [38] Workflows Best Practices - Cline
https://docs.cline.bot/features/slash-commands/workflows/best-practices