ChatGPT

Automating Daily Development Workflows with Cline CLI (Ask, Do, Compose Flows)

Cline CLI is an AI coding assistant that runs in your terminal and integrates into your development workflow. It supports three core usage **flows** for different scenarios: **Ask**, **Do**, and **Compose**. In an **Ask** flow, you interact with Cline in an **interactive chat mode** – Cline will propose a step-by-step plan (Plan Mode) which you can approve before execution [1] [2]. In a **Do** flow, you run Cline in **headless one-shot mode** (also called YOLO mode) to execute a task autonomously from start to finish with a single command [3]. And in a **Compose** flow, you create **multi-step workflows** (as Markdown files) that orchestrate complex or repetitive tasks by chaining commands, AI tools, and even multiple Cline instances in parallel [4] [5]. Below, we explore custom-built workflow examples for common software engineering activities – from code reviews to deployments – and show how to leverage each Cline flow effectively.

Pull Request Review Automation (Compose Flow)

Reviewing pull requests can involve repetitive steps that are ideal for automation. Using Cline's **Compose flow**, you can define a **PR Review workflow** that gathers context, analyzes changes, and even drafts review comments automatically [6]. For example, create a file `.clinerules/workflows/pr-review.md` in your repo with steps to:

1. **Gather PR Info:** Use the GitHub CLI to fetch the PR's title, description, and changed files (`gh pr view`) [7] [8].
2. **Examine the Diff:** Pull the unified diff of changes (`gh pr diff`) to let Cline see the code modifications [9].
3. **Analyze Changes:** Let Cline read the diff (and relevant files) and assess the changes for bugs, performance issues, or security concerns [10].
4. **Confirm Next Action:** Ask the user how to proceed – for example, present options to "Approve", "Request Changes", or "Comment" [11]. Cline uses an `<ask_followup_question>` prompt to pause for your choice.
5. **Execute Review:** Based on the choice, run the appropriate `gh pr review` command (approve, request changes, or comment with the prepared feedback) [12].

Once this **workflow** is defined, running it is simple. In the Cline CLI chat, trigger the workflow by typing its filename with the PR number as an argument, for example:

```
/pr-review.md 42
```

This single command launches the entire review sequence – Cline fetches the PR details, analyzes the diff, and presents findings, then executes the review action upon confirmation [13]. What used to take many manual steps is now handled in one go. (Ensure you have the GitHub CLI (`gh`) installed and authenticated for this workflow [14].) This PR review workflow showcases the power of **Compose flows**: we combined shell commands, file reading, and an interactive question into one automated process.

Automated Test Execution and Analysis (Do Flow)

Running test suites and interpreting results is another everyday task that Cline can streamline. In a **Do flow** (headless mode), you can ask Cline to run your tests and summarize the outcome autonomously. For example, you might execute:

```
cline task new -y "Run the full test suite and report any failures with
details."
```

Using the `-y` flag (YOLO mode) means Cline will plan and perform this task without any interactive prompts [15]. Under the hood, Cline can utilize its tools to run your test command and then analyze the output. In a custom workflow, this would be done with an `<execute_command>` step to run the tests, followed by having Cline parse the results. For instance, a workflow snippet could look like:

```
<execute_command>
  <command>npm run test</command>
  <requires_approval>false</requires_approval>
</execute_command>
```

Cline will execute the tests and capture the output for analysis [16]. After running the tests, the next workflow steps (or the autonomous agent in Do flow) can summarize which tests failed and why. Cline's AI capabilities allow it to read through logs or failure messages and produce a concise report of failing test cases or errors. In practice, a **Test Runner workflow** might output a summary like: "5 tests failed in module X: (list of failed test names with error messages) – likely cause: Y." This saves developers from combing through lengthy test logs. According to Cline's documentation, creating a workflow that runs your test suite and summarizes results is a common use case [17]. By automating test execution and triage in a Do flow, you can incorporate it into CI pipelines or nightly runs – for example, a Jenkins job could use Cline to run tests and **analyze failures, then even suggest fixes** for those failures [18].

Deployment Pipeline Automation (Compose Flow)

Deployments often involve a sequence of build, test, and release steps that can be encoded in a Cline workflow. Using the **Compose flow**, you can automate a deployment pipeline while still keeping human checkpoints for safety. Consider a **Staging Deployment workflow** (e.g., `deploy-staging.md`) that performs the following steps:

1. **Run Tests** – Execute the full test suite to ensure the build is sound (`npm test`) [19].
2. **Build Artifacts** – Build the application or create production artifacts (`npm run build`, or compile binaries, etc.) [19].
3. **Deploy to Staging** – Use CLI tools to deploy to a staging environment (for example, using Docker/Kubernetes commands like `docker compose up` or `kubectl apply`) [19].

4. **Health Check** – After deployment, run a quick health check (e.g., `curl http://staging.myapp/health`) to verify the service is up ⑲ .
5. **Notify Team** – Integrate with Slack to post a deployment notification. Cline can use a Slack **MCP** integration to send a message (e.g., “Staging deployment for version X is complete ”) to your team’s channel ⑲ .
6. **Await Approval for Production** – Use an `<ask_followup_question>` to pause and ask the engineer: “Do you want to deploy to production now?” with **Yes/No** options ⑳ ㉑ . If “Yes”, the workflow could then proceed to run the production deployment steps; if “No”, it can halt or schedule for later.

Such a workflow encapsulates an entire release process. When you trigger it (e.g. `/deploy-staging.md` in the Cline interface), Cline will carry out each step in order ㉒ . Crucially, this automation still keeps you in the loop – by notifying the team and requiring explicit approval for production, you maintain control over critical decisions. This aligns with best practices: *“multiple tools (CLI commands, file operations, MCP servers, user interaction) orchestrated into a single command”* is exactly what Cline workflows enable ㉓ . Integrations shine here as well: you can connect a Slack MCP or other service integrations so that your deployment workflow not only runs shell commands but also interacts with external systems. Cline’s documentation notes that deploying applications using tools like Docker/Kubernetes is a common automation scenario ㉔ . By scripting it in a Compose flow, you ensure every deployment is consistent and nothing is forgotten (no manual steps), which is ideal for CI/CD. In fact, you could even trigger Cline deployment workflows via CI – for example, a GitLab pipeline could call `cline` to deploy a feature branch after tests pass ㉕ .

Code Documentation Generation (Do Flow)

Maintaining up-to-date documentation in code (like function docstrings or API docs) is another tedious task that can be accelerated with Cline. In a **Do flow** one-liner, you can have Cline add or update documentation throughout your codebase. For instance, to generate JSDoc comments for all functions in the `src/` directory, simply run:

```
cline task new -y "Add JSDoc comments to all functions in src/"
```

Cline will autonomously search through the code, open each file, and insert appropriate documentation comments where they’re missing ㉖ . Because Cline’s agent can read and write files, it will ensure the comments are contextually correct (describing function parameters, return values, etc.). This is done in Plan-and-Act under the hood: Cline plans which files and functions need docs, then in Act mode it edits those files. The `-y` flag makes it fully automatic. It’s recommended to run such documentation generation on a clean git branch or review the changes, since YOLO mode trades oversight for speed ㉗ . Beyond inline code comments, you could create workflows for broader documentation tasks – e.g. generating or updating Markdown docs. For example, a workflow might use `read_file` to gather content from code or design docs, then `write_to_file` to compose an updated `README.md` or API reference. Cline’s `write_to_file` tool can be used to create documentation files from templates or scratch ㉘ ㉙ . In sum, the AI can handle the heavy lifting of documentation, and you invoke it with a simple command. This saves time and ensures consistency in how docs are written across the project.

Bug Triage and Fix Workflows (Ask/Compose Flow)

When a new bug is reported, Cline can assist in triaging and even fixing it through a mix of interactive and scripted flows. In an **Ask flow**, you might start an interactive Cline session to diagnose the issue. For example, you could provide the error log or bug description to Cline in chat and ask, *"What could be causing this bug and how do I fix it?"*. Cline will enter Plan Mode to investigate: it might search the codebase for the error message or relevant modules using the `search_files` tool [30], read the identified source files (`read_file` tool) to pinpoint the faulty code, and then propose a fix plan. You can refine this plan through conversation and then let Cline apply the fix in Act Mode. This human-in-the-loop approach is useful when the bug is complex or you want to supervise each step of debugging.

For recurring bug triage processes, you can also create a structured **Compose workflow**. For example, a **"Bug Fix" workflow** might outline steps similar to a mini playbook:

1. **Reproduce the Bug:** If possible, run the application or tests to trigger the bug (or have Cline generate a failing unit test that demonstrates the bug) [31]. This step ensures the problem is well-understood and will be solved by the fix.
2. **Investigate:** Use logging or debugging tools to gather more insight. A workflow could insert temporary log statements or run the program with debug flags. Cline might utilize `<execute_command>` to run the app/test and then parse any stack trace or error output. It can also search the code for suspects (e.g., look for the function named in the error) to focus the investigation.
3. **Apply Fix:** Once the cause is identified, Cline can modify the code to fix the bug. In autonomous mode, it will directly edit files (`write_to_file` operations) to implement the minimal change needed [31]. In interactive mode, it might show you the diff of the proposed change for approval (you can always ask Cline to explain its changes before proceeding).
4. **Verify and Test:** After the fix, the workflow should re-run the relevant tests or reproduction steps to confirm the bug is resolved. This can be automated by another `<execute_command>` running the test suite again [32]. Cline will check that the previously failing test now passes and that no new regressions are introduced.
5. **Document and Commit:** Finally, Cline can add a comment in the code explaining the fix or update a CHANGELOG. It could even draft a commit message (e.g., "fix: [issue description] (closes #123)") following your conventions [32]. You could integrate with issue trackers here – for example, update the status of the corresponding Jira ticket via an API call or MCP server once the fix is confirmed.

Such a workflow encapsulates the **bug triage** lifecycle from identification to resolution. Integrations with external tools make it even more powerful: Cline's Model Context Protocol (MCP) allows connecting to services like Jira or GitHub. This means your workflow could automatically fetch a bug report from Jira, or post a comment on a GitHub issue when the fix is ready [33]. In practice, you might use a Jira MCP server (for example, the open-source `jira-cli-mcp` [34]) so that Cline can query ticket details or update an issue's status as part of the workflow. Slack integrations can also broadcast a message to your team (e.g., "Bug XYZ has been fixed and deployed to staging") to close the loop. By leveraging Cline for bug triage, engineers can reduce the toil in reproducing issues and ensure no step is overlooked in the fix process.

Integrating Cline with Your Toolchain (GitHub, Jira, Slack, VS Code)

One of Cline's strengths is how well it integrates with existing developer tools and workflows. You can trigger Cline workflows in your **CI/CD pipeline** or from other automation tools. For example, it's possible to add Cline to a GitHub Actions workflow so that every pull request is automatically analyzed by the AI agent [18]. Cline can be used in GitLab or Jenkins jobs to generate code artifacts (like migration scripts) or to analyze test failures and provide feedback in an automated fashion [18]. All of this is done via the CLI – e.g., running `cline task new -y "Analyze the latest test failures and suggest fixes"` as a step in your CI script.

For source control integration, using the **GitHub CLI** (`gh`) within Cline workflows (as shown in the PR review example) is a straightforward way to interact with repositories and PRs. Cline can also interface with Git operations directly; there are MCP servers that give Cline more direct Git superpowers if needed [35], but often invoking standard CLI tools (git, gh) in workflows is sufficient.

To integrate with issue trackers like **Jira**, or communication platforms like **Slack**, Cline supports **MCP plugins**. The Model Context Protocol allows Cline to execute actions on external services as if they were part of its toolset. For instance, by enabling a Slack MCP, your workflow can send messages or respond to Slack events; with a Jira MCP, it can create or update Jira issues [33]. The Cline community provides many such integrations (for databases, cloud platforms, etc.), and you can configure them in Cline's settings. An example from Cline's docs shows how adding an MCP server for GitHub or Slack is as easy as specifying the server in a config JSON with the appropriate credentials, after which Cline can call those APIs securely [36] [37]. This means your custom workflows can reach beyond your local environment – they can trigger builds on AWS, post updates to Slack, query a database, or anything else for which an integration exists.

Finally, Cline works wherever you work. There is a **VS Code extension** and other editor integrations that tie into the same Cline backend. In fact, multiple frontends (the CLI, VS Code, JetBrains IDEs, etc.) can attach to the same running Cline instance [38]. This allows seamless hand-off: you might start an interactive Ask flow in VS Code's Cline chat panel, then switch to the terminal and continue the task with `cline task chat` or vice versa. The workflows and rules you develop are shareable across your team (since you can commit the `.clinerules/workflows` directory to version control [39] [40]) and portable across environments. Whether triggered from a chat command in your IDE or a script in your CI, Cline's automation flows help ensure daily development tasks are handled consistently and intelligently.

Sources:

- Cline Documentation – *Overview & Three Core Flows* [1] [3]
- Cline Documentation – *Workflows Quick Start (PR Review Example)* [7] [11]
- Cline Documentation – *Workflows Best Practices & Tools* [16] [20]
- Cline Official Blog – *One-Shot Automation with Workflows (v3.16)* [4] [6]
- Cline Official Blog – *Stop Adding Rules, Use Workflows* [10] [19]
- Reddit – *MCP Integrations (Slack, GitHub, Jira)* [41] [42] (Real-user insight on useful integrations)
- LinkedIn – *Practical Guide to Cline (Custom Workflow Examples)* [31] [33] (Community examples of bug-fix workflow and MCP usage)

---

① ② ③ ⑮ ㉖ ㉗ Three Core Flows - Cline
https://docs.cline.bot/cline-cli/three-core-flows

④ ⑥ Cline v3.16: One-Shot Automation with Workflows, Plus UI & Stability Gains
https://cline.ghost.io/cline-v3-16-one-shot-automation-with-workflows-plus-ui-stability-gains/

⑤ ⑩ ⑲ ㉒ ㉓ Stop Adding Rules When You Need Workflows
https://cline.ghost.io/stop-adding-rules-when-you-need-workflows/

⑦ ⑧ ⑨ ⑪ ⑫ ⑬ ⑭ ⑰ ㉔ Workflows Quick Start - Cline
https://docs.cline.bot/features/slash-commands/workflows/quickstart

⑯ ⑳ ㉑ ㉘ ㉙ ㉚ ㊴ ㊵ Workflows Best Practices - Cline
https://docs.cline.bot/features/slash-commands/workflows/best-practices

⑱ ㉕ Overview - Cline
https://docs.cline.bot/cline-cli/overview

㉛ ㉜ ㉝ ㊱ ㊲ Supercharge Your Development Workflow: A Practical Guide to Cline and Claude
https://www.linkedin.com/pulse/supercharge-your-development-workflow-practical-guide-kartik-patel-gcxpc

㉞ choplin/jira-cli-mcp: MCP server that wraps jira-cli to enable ... - GitHub
https://github.com/choplin/jira-cli-mcp

㉟ ㊶ ㊷ After testing 20+ MCPs in Cline, only these few actually changed my workflow. Here’s why : r/CLine
https://www.reddit.com/r/CLine/comments/1nbrxrs/after_testing_20_mcps_in_cline_only_these_few/

㊳ CLI Reference - Cline
https://docs.cline.bot/cline-cli/cli-reference