ChatGPT

Using Claude Code (via Z.AI) for Rewriting a Rust + React Web App

Claude Code is an agentic coding assistant that runs in your terminal and integrates with the Z.AI platform (which offers Anthropic's GLM-4.5 models). It can read and modify your codebase through natural language commands, helping you rewrite and refactor a large project. However, working on a **Rust + React web application** with **large context** (long documentation, many large files) requires careful strategies to avoid context overflow and minimize errors. Below is a comprehensive guide on setting up Claude Code, handling long contexts, understanding architecture, and using advanced tricks (from configuration settings to grep and prompt techniques) to successfully rewrite an existing large project with minimal mistakes.

Setting Up Claude Code with Z.AI Integration

Before diving into usage, ensure Claude Code is properly installed and configured to use Z.AI's API (GLM-4.5 models):

- **Installation:** Claude Code is distributed via NPM. Install it globally and launch it in your project folder [1]. For example:

```
npm install -g @anthropic-ai/claude-code
cd your-project-directory
claude
```

- **Z.AI API Key:** Obtain an API key from Z.AI and configure Claude Code to use the GLM-4.5 model through Z.AI's endpoint [2] [3]. This typically involves setting environment variables:

```
export ANTHROPIC_BASE_URL="https://open.bigmodel.cn/api/anthropic"
export ANTHROPIC_AUTH_TOKEN="your_zhipu_api_key"
```

- **Model Configuration:** By default, Claude Code uses two models in tandem for efficiency [4]:
- `GLM-4.5` (full model) for dialogue, planning, coding, and complex reasoning.
- `GLM-4.5-Air` (faster, lightweight model) for auxiliary tasks like file searching and syntax checking.

This hybrid setup provides more tokens and reliability at a lower cost [5] [6]. You typically don't need to switch models manually – Claude Code will automatically delegate tasks to the appropriate model. (Currently, switching to other models is not supported [7].)

- **Launching and Permissions:** The first time you run `claude` in your project, grant it permission to access your files [8]. Claude Code will request confirmation before executing potentially risky operations (like editing files or running commands). You can streamline this by customizing the allowlist (more on this later).

With setup done, you're ready to leverage Claude Code on your Rust + React codebase. The following sections cover how to manage large code contexts, understand and refactor the architecture, and use advanced techniques to minimize errors.

Gaining an Architectural Overview of the Project

When rewriting a large web application, it's crucial to first understand the **existing architecture and code structure**. Claude Code can help you quickly get up to speed on a new or inherited codebase:

- **Overview and Architecture Queries:** Simply ask Claude for a high-level summary of the codebase. For example, you can prompt: *"Give me an overview of this codebase."* Claude will autonomously scan and summarize the project's structure and key components [9]. Follow up with specific architecture questions, such as: *"Explain the main architecture patterns used here."* or *"What are the key data models in this project?"* [10]. Claude will identify relevant files and patterns to answer these questions.

- **Focused Questions:** Dive deeper by asking about particular subsystems. For instance: *"How is authentication handled?"* [11] or *"Trace the login process from front-end to database."* These targeted queries prompt Claude to traverse the connections between front-end React components, backend Rust services, and database code [12]. It uses its agentic abilities to find the relevant code across the repository and describe how they interact.

- **Finding Relevant Code:** If you need to locate where a certain feature or function is implemented, you can literally ask Claude to find it. For example: *"Find the files that handle user authentication."* Claude will search your project and list the files related to that feature [13]. This acts like an AI-powered `grep`, sparing you manual searching. Once you have the list, you can ask: *"How do these authentication files work together?"* to understand the flow [14].

- **Project Documentation via CLAUDE.md:** Claude Code supports **memory files** (persistent context). One special file is `CLAUDE.md` placed at the project root (or other locations) which is automatically pulled into context on startup [15] [16]. You can use this to document the project's architecture, important conventions, or any notes the AI should always remember. For example, you might include a summary of each module, key design decisions, or coding standards. Keeping this file **concise and informative** improves Claude's understanding of the overall system [17]. (Claude also loads organization-wide and user-specific CLAUDE.md files if present, merging instructions hierarchically [18] [19].)

- **Automated Architecture Notes:** If the project is extremely large or "spaghetti code," consider using Claude to **generate documentation** for sub-parts of the system. One Reddit user suggests having Claude "untangle the modules" by producing a `how-it-works.md` for each module, including all function signatures and notes on quirks [20]. This essentially compresses each part of the codebase into a human-readable explanation. After generating such module-wise docs, you (or Claude) can then synthesize a higher-level understanding of the entire project from them [20]. This approach helps when the code is too unwieldy to load all at once – you create intermediate summaries that fit in context.

Tips: Start with broad questions and progressively narrow down [21]. Ask Claude to define project-specific terminology or acronyms it encounters – building a quick glossary can clarify things [21]. All these steps give you and Claude a shared understanding of the architecture before you attempt major changes.

Strategies for Long Contexts and Large Files

Working with a **large codebase** means you'll quickly hit context length limits if you naively try to load whole files or multiple files at once. Even though Claude Code's model (GLM-4.5) supports a very large context window (up to around 200k tokens as reported) [22], a sprawling project may still exceed this. Here are strategies to handle long contexts and large files:

- **Agentic Code Search:** Claude Code does **not** blindly stuff your entire repository into the prompt for every query. Instead, it intelligently finds relevant snippets and files autonomously [23] [24]. It might use internal filtering (like keyword search or embeddings) to decide which parts of the code to read in order to answer your query. This means you should focus your questions to guide its search effectively. As one user notes, *"you have to be precise with your prompt and guide it [on] what you really want"* [25]. The better you scope the task or specify filenames, the less wasted context.

- **Explicitly Provide Key Context:** Whenever possible, **directly provide the relevant code** to Claude instead of relying wholly on its search. For example, if you know you need to refactor a specific function or file, open or copy that code into the conversation. *"Ideally you want to quote the relevant, exact file(s) or methods in the prompt,"* advises one experienced user [26]. By including only the necessary code sections, you avoid hitting token limits with unrelated text and reduce the chance of errors.

- **Plan Mode for Exploration:** Claude Code offers a **Plan Mode** for safe, read-only analysis of your codebase. In Plan Mode, Claude will **analyze extensively without making changes** [27]. This is useful when you need a comprehensive understanding or a step-by-step plan that spans many files. For instance, before implementing a broad refactor, you can activate Plan Mode and ask: *"Analyze the authentication system and suggest improvements."* Claude will scan relevant parts of the codebase to draft a plan [28]. Plan Mode is recommended for multi-step implementations or when exploring unfamiliar code, because it ensures Claude won't modify anything while gathering context [29]. You can enter Plan Mode by pressing **Shift+Tab** twice during a session (you'll see `⏸ plan mode on` indicator) or start a session with `claude --permission-mode plan` [30]. Use this to compile information and a strategy from across the project *within* the model's thinking process, then review the plan before execution.

- **Memory and Context Trimming:** Even with intelligent search, long sessions can accumulate a lot of conversational history and file content in the context window. Claude Code allows you to clear or trim context when needed. Use the `/clear` command to reset the conversation history between distinct tasks [31]. This is important to **prevent irrelevant or stale information** from consuming the context and confusing the model. For example, after finishing one chunk of the rewrite, clear the context before moving to the next module so that old file contents don't linger. Additionally, keep your prompts focused on the current task; avoid lengthy tangents in one session.

- **Splitting Large Files:** If you have extremely large files (e.g. a single 18,000-line React component or a huge Rust module), Claude Code can handle them better than many tools [32], but you might still approach them in sections. You can ask for a summary of a large file first, or use Claude to extract the parts that need changes. In cases where a file is too big, consider using your editor or command-line to split out relevant portions. Claude Code's ability to update very large files is quite robust – in fact, an 18k-line React file was successfully updated by Claude Code where other agents failed [32] – but it's wise to tackle big files methodically (e.g., function by function or segment by segment) if errors start to creep in.

- **Feeding External Data (Grep Tricks):** If Claude's built-in search isn't pinpointing what you need, you can run your own search (like `grep` or `rg` on the command line) and feed the results. Claude Code supports **piping data into its input** [33]. For example, you might do:

```
grep -R "OldFunctionName" . > grep_results.txt
cat grep_results.txt | claude
```

Then ask Claude to analyze those results. Alternatively, simply copy-paste a snippet of `grep` output into Claude's prompt (be sure to format it clearly, e.g., "Output of `grep` for 'OldFunctionName': ..."). Claude can read that and help determine where changes are needed. Claude Code can also execute shell commands if permitted; you could instruct it to run a grep itself via a Bash command (after adding `grep` to allowed tools or in a controlled prompt). Always be cautious with very large outputs though – filter the grep results to manageable size before feeding them.

- **Using `CLAUDE.md` Imports:** As mentioned earlier, you can use the `CLAUDE.md` mechanism to preload essential context. This can include imports of other documentation files. For instance, you might import your project's README or design docs into the Claude memory using an import line like `See @README.md for project overview` [34]. These imported files (up to a certain depth) will be automatically included in Claude's context on startup [35]. It's a way to give the model a *head start* on understanding the project without manually copying that info every time. Make sure any such docs are concise or consider splitting them, because they will consume tokens every session.

By combining these tactics – guiding Claude with precise prompts, leveraging its plan mode and memory features, and manually trimming or feeding context where needed – you can effectively work within context limits and ensure Claude always has the **most relevant information** for the task at hand.

Claude Code Settings and Advanced Tooling Tricks

Claude Code is highly configurable. Adjusting its settings and knowing its toolset will improve efficiency and reduce friction during development:

- **Project and User Settings:** Claude Code reads configuration from JSON files (like `~/.claude/settings.json` or a project-specific settings file). You can set default behaviors here. For example, you might configure the default permission mode or allowed tools. To always start in Plan Mode by default (if you find yourself using it often), you could add to settings:

```
{
  "permissions": { "defaultMode": "plan" }
}
```

which ensures every new session begins in plan mode [36]. Similarly, you can preset an allowlist of tools or commands your project uses frequently.

- **Skipping Permissions Prompting:** By design, Claude Code asks for confirmation before performing actions like editing a file or running a shell command, as a safety measure [37]. This can become tedious when you trust Claude and need to automate changes. One trick (for advanced users in a safe environment) is to launch Claude with the `--dangerously-skip-permissions` flag [38]. This will let it execute edits and allowed commands without pausing for approval every time. *Use this with caution* – review what Claude is about to do, especially for destructive operations. In practice, many find it speeds up development significantly and outright harmful actions are rare [38]. Alternatively, you can gradually build an **allowlist**: as prompts come up, choose "Always allow" for specific tools (like file `Edit` or `Bash(cargo *)` commands) [39] or use the `/permissions` command to edit permissions mid-session [39].

- **Custom Tools and Hooks:** Claude Code can be extended with **slash commands and hooks**. For instance, if you frequently need to run a certain analysis (like running Rust's Clippy linter or a specific `grep` pattern), you could create a custom slash command that Claude can execute on demand. Claude can even help you write these hooks/commands – you can ask it to generate a hook script, and place it in the `.claude/` directory for use [40]. Setting up hooks like a PreToolUse or PostToolUse script allows you to automate repetitive tasks (e.g., maybe automatically format code after editing, or log each change). While developing your rewrite, this is optional, but worth noting for complex workflows.

- **Using Multiple Claude Instances (Parallel Agents):** For very large projects, consider running more than one Claude Code session in parallel. Each instance works on an isolated context (especially if you use separate Git worktrees or folder copies of your repo) [41] [42]. This is useful if you want to **divide tasks** – for example, one Claude working on backend Rust code and another on the React frontend simultaneously. Anthropic engineers often use 3-4 parallel Claude sessions in separate terminal tabs, each on a different sub-project or feature branch [43]. If you do this, ensure you coordinate via version control (Git) to merge changes later. You could even have one instance focus on generating code, and another strictly for reviewing or testing (discussed more below) [44].

- **Extended "Thinking" with Checklists:** If you have a very complex refactor or a task that touches many parts of the code, instruct Claude to use a **scratchpad or checklist**. For example, say you need to update dozens of API endpoints or fix numerous lint errors. You can prompt Claude to output a markdown checklist of all items to address (perhaps by running a diagnostic command and capturing its output) [45]. Claude can list all the places requiring changes, then proceed to tackle them one by one, checking off each item as it's completed [46]. This structured approach prevents context overload because Claude focuses on one sub-task at a time while keeping the big picture list for reference. It's effectively forcing the model to **plan and execute stepwise**, which reduces omissions and errors in large-scale changes.

- **Integrating External Tools:** Claude Code can leverage external commands via Bash. Use this to your advantage. For instance, when working with Rust, you might frequently compile or run tests. You can simply tell Claude in natural language: *"Run the test suite"* or *"Compile the project"*, and it will translate that to the appropriate shell command (`cargo test` or `cargo build`) if those are allowed. After running, it will read the output and incorporate it into context. Make sure to allow those commands (e.g., allow `Bash(cargo * )` in permissions). This ability to **pull in real-time data** means Claude can catch errors by seeing compiler or test output and then help fix them [46] [47]. The Z.AI Claude integration also supports **MCP (Model Context Protocol) tools** – for example, if you have the web browsing or vision tool enabled (less likely needed for a coding task), Claude can use them. But the most relevant "tools" in coding are your compiler, test runner, linter, etc., which you should not hesitate to have Claude invoke.

- **VS Code / IDE Extension:** If you prefer a GUI, note that Claude Code can be launched via a VS Code extension [48]. This isn't mandatory, but the extension simply helps you open Claude in an IDE pane and manage multiple instances. It doesn't change Claude's capabilities (it's essentially the same CLI agent in the background), but can be convenient to visualize changes side-by-side with your code.

In summary, tweak Claude Code's environment to suit your workflow. By reducing permission friction, giving it helpful custom commands, and possibly splitting workloads across multiple agents, you'll make the rewrite process faster and smoother.

Effective Prompting for Code Generation and Refactoring

Writing good **prompts** is key to getting high-quality, low-error code from Claude. Here are prompt strategies and examples specifically useful for Rust/React development and large-scale refactoring:

- **Be Specific and Incremental:** The more specific your instruction, the less room for error. Rather than saying *"Rewrite this project in Rust and React"* in one go (far too broad!), target one component or module at a time. For example: *"Convert the user authentication module from Node.js to Rust (using Actix Web on backend). These are the current behaviors... Ensure the React frontend calls the new Rust API correctly."* If there's a particular file or function to focus on, mention it by name. This helps Claude fetch just the relevant context. Users report that Claude excels when you *guide it step-by-step* through complex tasks [25], rather than issuing an overly general command.

- **Use Role-Playing or Perspective in Prompts:** Sometimes it helps to invoke Claude's knowledge of best practices by assigning it a role. For instance: *"You are a Rust expert and React architect. Please refactor the following code to improve performance and readability."* While Claude Code's system prompt already includes a coding assistant persona, reinforcing context like the target tech stack can focus it. You might prompt: *"Rewrite the below Python logic in Rust. Make sure to handle errors idiomatically with* `Result` *and use appropriate Rust crates."* By specifying these details, you reduce the chance of non-idiomatic output.

- **Incorporate Documentation in Prompts:** If rewriting from another language or framework, you may need to consult documentation (for example, an existing system's API or a library's usage). You can paste snippets of docs and then ask Claude to implement according to that spec. Claude's large context window allows inclusion of doc fragments (just be mindful of the token limit). Also, if

rewriting in Rust, consider including relevant parts of the Rust crate documentation or interface definitions for reference. Claude can then align its output to those definitions, minimizing mistakes.

- **Prompt to Generate Tests or Types First:** A neat trick when rewriting is to first ask Claude to generate type definitions or tests, then implement the code to satisfy them. For instance, *"Define the TypeScript types for the data structures used between the Rust backend and React frontend."* Once you have those, it becomes easier to prompt Claude to implement the Rust data model and the matching TypeScript interfaces without inconsistencies. Or, *"Write a Rust unit test (or React component test) for the desired behavior of X,"* then have Claude implement the code to make that test pass. This test-driven prompt approach can catch errors early.

- **Leverage "Extended Thinking" in Prompts:** If Claude's response seems incomplete or if it leaves `TODO` comments (indicating it wasn't sure about something), prompt it to think more. For example, *"Break down the solution step by step before writing code,"* can encourage Claude to outline a plan (like pseudocode or a list of sub-tasks) within its answer. Claude Code has a concept of "Extended thinking" or using a scratchpad: you can explicitly ask it to reason or list assumptions. This often leads to more correct and thorough code because Claude works through the problem in the open. In the CLI, using Plan Mode as described is one way to enforce this planning phase. Even outside Plan Mode, you can ask for a plan first: *"Plan the changes needed to migrate module X to Rust, list each file and function to change."* Once it lists them, you then say *"Great, now implement step 1."* and so on. This guiding technique keeps the context relevant and helps the model avoid going off-track.

- **Iterate and Refine:** Don't expect a perfect solution on the first try. Claude might produce an initial version of a Rust function that doesn't compile or a React component with small bugs. This is normal. Your prompt cycle should include *reviewing the output, testing it, and feeding back errors* for Claude to fix. For example, after Claude writes a Rust module, try compiling it (either yourself or instruct Claude: *"Run* `cargo check` *."*). If errors are found, copy the compiler error messages back into Claude: *"Here are the compile errors: ... How can we fix these?"* Claude will analyze the error messages and correct its code accordingly. It's adept at debugging its own output when given clear error logs or test failures. In fact, a best practice is to *share the stack trace or error text directly* with Claude and ask for a fix [49] [50] .

- **Use Examples and Constraints:** If you have a particular coding style or you want to avoid certain pitfalls, include that in your prompt. For instance: *"Here is an example of how we handle state in our Rust code (provide a short code snippet). Please follow this style when implementing the new module."* Or *"Avoid using global variables; pass dependencies via function arguments."* Constraints like these can be put in the `CLAUDE.md` as well so that they're always considered [51] . Emphasize critical points (Anthropic suggests using words like "IMPORTANT: ..." or even all-caps for must-follow rules in the memory or prompt [52]) to ensure Claude adheres to them.

Remember, Claude Code is quite **robust and "intelligent" in parsing your intent**, but it's not infallible. Clear and contextual instructions will reduce the chance of errors or misinterpretation significantly.

Iterative Development and Minimizing Errors

One of the goals is to rewrite the project with **minimum errors**. Here's how to ensure quality at each step of using Claude Code:

- **Small Commits / Checkpoints:** Treat each Claude Code operation as a small commit. After it makes changes, review the diff (Claude will show what it edited). Verify that the change is correct. It's easier to catch mistakes if you change 2-3 files at a time versus 20 files in one go. You can ask Claude to summarize what it did: *"Summarize the changes you just made."* This helps double-check it didn't do something unintended. Using version control (git) is highly recommended – commit after each successful set of changes. If something goes wrong, you can revert or use `git diff` to pinpoint the issue.

- **Testing Early and Often:** Leverage your test suite if one exists. After Claude rewrites a component or module, run its tests (or have Claude run them for you). If tests fail, share the failing test outputs with Claude. For front-end React components, you might run your build or use a tool like React Testing Library. For Rust, definitely run `cargo test`. Claude can fix logic errors or adjust its output based on test feedback. As the docs suggest, ask for both unit and integration tests where appropriate [53] – you can even have Claude *generate new tests* for critical functionality and then verify the code against them [54] [55].

- **Code Reviews with Claude:** A powerful pattern is to use **multiple instances of Claude in a review cycle**. For example, after one Claude (let's call it the "coder") produces some code, you can copy that code (or commit it) and then in another terminal start a fresh Claude session (the "reviewer") that is not biased by the prior conversation. Ask the reviewer Claude: *"Review the following code for any errors or improvements,"* and paste the code. This second instance will analyze it critically and might catch mistakes or suggest better approaches [44] [56]. You can then take those suggestions back to the coder instance or apply them yourself. This separation of concerns – one AI writes, another checks – often yields higher quality results [44], similar to a pair programming or code review scenario.

- **Static Analysis and Linting:** Use Rust's compiler as a static analyzer – it's very strict, which is good. Also use `rustfmt` and `clippy` (Rust linter) to catch style issues or common gotchas. You can have Claude run `cargo clippy` and interpret the warnings. For JavaScript/TypeScript (React side), use ESLint or `tsc` for type checking. These tools can be integrated into the Claude workflow: *"Run ESLint on the frontend code and show me any issues,"* then address them one by one (Claude can fix lint issues systematically – as noted in the tips, you can even automate it via a checklist [57] [46]). Combining AI with traditional linters combines the strengths of both.

- **Use Plan-Implement-Review Loop:** Summarizing the above into a recommended workflow for each chunk of the rewrite:

- **Plan:** In Plan Mode or via a planning prompt, outline what needs to be done (which files/functions to change, what new code to write). Ensure the plan makes sense and covers all requirements.
- **Implement:** Switch to normal mode (or auto mode) and have Claude carry out the changes as per plan. Monitor the changes it's making (Claude will typically present diffs or describe the edits).

- **Test/Review:** Immediately test that piece (run code, execute tests, or spin up the dev server). Also consider using a second Claude instance or simply scrutinize the diff for anything suspicious. If issues are found, loop back: feed Claude the error or comment and let it fix, or manually tweak and inform Claude of the changes so it stays in sync.

- **Clear Context:** Once that task is done and verified, use `/clear` to wipe the slate (or start a new session for the next task). This prevents any irrelevant context from bleeding into the next implementation.

- **Documentation and Comments:** As you rewrite, ensure to update documentation and inline comments with Claude's help. Ask Claude to add docstrings or comments explaining complex parts of the new code. This not only validates that Claude *itself* understands the code it wrote (helping catch logical errors if the explanation doesn't align with the code), but leaves you with a well-documented codebase. You can prompt: *"Add JSDoc comments to the undocumented React functions in this module"* [58] or *"Document the new Rust API endpoints with comments explaining their behavior."* Good documentation will make future maintenance easier and can be done as part of the rewrite process.

- **Keep Claude Updated on External Changes:** If you manually edit some code outside of Claude (or if one Claude instance made changes that another isn't aware of), make sure to **sync Claude's view**. You can do this by opening the file in Claude or telling it what you changed. Claude Code doesn't magically know about edits done outside of its session, so to avoid it introducing regressions, confirm it's working on the latest version of the file. A simple way is to open the file in Claude and scroll (Claude will load it), or use the `/reload` command if available. Consistency is key to avoiding errors.

In practice, Claude Code can produce production-ready Rust and React code, as long as you guide it and verify each step. Users have found that *"Anthropic definitively makes the best coding models"* for complex tasks [59], and Claude Code's design (with planning capabilities and tool use) makes it reliable even for large files and intricate refactors [32] [60]. It rarely gets stuck or needs babysitting, compared to other coding assistants [60], so you can trust it with quite challenging tasks. Just remember that **your oversight and iterative feedback are crucial** to achieving a correct and efficient rewrite.

Conclusion

Rewriting a large web application in Rust and React with Claude Code is an achievable goal when you harness the tool effectively. Start by understanding the existing system through Claude's eyes – get overviews and identify key pieces. Use memory files and project documentation to give Claude the necessary context. Tackle the rewrite in well-defined chunks, using Plan Mode to layout steps and normal mode to execute them. Always keep an eye on the context window: feed in only what's needed (prune irrelevant info, use grep or targeted file queries to focus the AI). Employ Claude Code's settings and features – whether it's customizing `CLAUDE.md` instructions, automating tasks with checklists, or even spinning up parallel Claude instances – to supercharge your workflow. Throughout, validate the AI's output via tests, compilers, and secondary reviews to catch errors early.

With these practices, Claude Code becomes a powerful pair-programmer that can handle lengthy documentation and sprawling codebases. It will help map out your project's architecture, suggest

improvements, and actually implement large-scale changes across Rust backend code and React frontend code. By iterating diligently and leveraging all "tricks" at your disposal (from context management to prompt engineering), you can significantly **reduce errors** and complete the rewrite faster than with manual effort alone. Claude will not only follow your instructions but also provide insights and fixes along the way – truly enabling you to *"code faster, debug smarter, and manage workflows seamlessly"* even in the face of long docs and large files [5].

Sources:

- Anthropic Claude Code Documentation – *Installation and Z.AI Integration* [1] [3]; *Hybrid Model Usage* [4]; *Project Overview Query* [9]; *Architecture Q&A* [10] [12]; *Finding Relevant Files* [13]; *Error Fixing Workflow* [49] [50]; *Plan Mode Explanation* [27] [28]; *Documentation and Comments* [58]; *Memory (CLAUDE.md) Best Practices* [15].

- Anthropic Claude Code Best Practices – *Using /clear for Context* [31]; *Checklists for Large Tasks* [61]; *Feeding Data via Pipe* [33]; *Parallel Claude Instances & Worktrees* [41]; *Multi-Claude Review Workflow* [44]; *Skip Permissions Flag* [37].

- Reddit (ClaudeAI community) – *Handling Context Limits* [26] [25]; *Strategy for Spaghetti Code* [20].

- Builder.io Engineering Blog – *Claude Code with Large Files & Reliability* [32] [60].

---

[1] [2] [3] [4] [5] [6] [7] [8] Claude Code - Z.AI API DOC
https://docs.z.ai/devpack/tool/claude

[9] [10] [11] [12] [13] [14] [21] [27] [28] [29] [30] [36] [49] [50] [53] [54] [55] [58] Common workflows - Anthropic
https://docs.anthropic.com/en/docs/claude-code/common-workflows

[15] [16] [17] [31] [33] [39] [41] [42] [43] [44] [45] [46] [47] [51] [52] [56] [57] [61] Claude Code Best Practices \ Anthropic
https://www.anthropic.com/engineering/claude-code-best-practices

[18] [19] [34] [35] Manage Claude's memory - Anthropic
https://docs.anthropic.com/en/docs/claude-code/memory

[20] [22] [25] [26] How does Claude Code manage large codebases beyond its context limit? : r/ClaudeAI
https://www.reddit.com/r/ClaudeAI/comments/1leclvr/how_does_claude_code_manage_large_codebases/

[23] [24] Claude Code's Context Magic: Does It Really Scan Your Whole Codebase with Each Prompt? : r/ClaudeAI
https://www.reddit.com/r/ClaudeAI/comments/1jpyqlu/claude_codes_context_magic_does_it_really_scan/

[32] [37] [38] [40] [48] [59] [60] How I use Claude Code (+ my best tips)
https://www.builder.io/blog/claude-code