Prompt Engineering for AI Agents

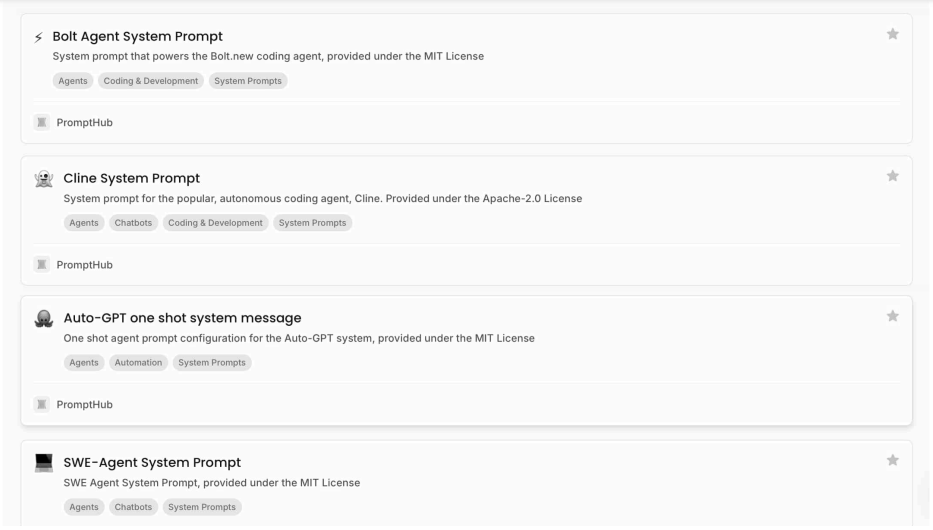
Last updated on August 12, 2025

Contents

2025 seems to be shaping up as the year of the AI agent. From coding to customer support, models seem to now be good enough to perform tasks agentically.

In this guide, we'll dive into what prompt engineering for agents really means and how it differs from non-agentic situations. One of the best ways to learn is by seeing what other teams are doing. We've pulled together a collection of 20+ prompts from popular open-source agents like Bolt.new, Cline, and more.

View transcript


Bolt Agent System Prompt
System prompt that powers the Bolt.new coding agent, provided under the MIT License
Agents · Coding & Development · System Prompts
PromptHub

Cline System Prompt
System prompt for the popular, autonomous coding agent, Cline. Provided under the Apache-2.0 License
Agents · Chatbots · Coding & Development · System Prompts
PromptHub

Auto-GPT one shot system message
One shot agent prompt configuration for the Auto-GPT system, provided under the MIT License
Agents · Automation · System Prompts
PromptHub

SWE-Agent System Prompt
SWE Agent System Prompt, provided under the MIT License
Agents · Chatbots · System Prompts

Check out the full collection here

What are agents?

Many people, myself included, have at one point or another mischaracterized what an agent is. This is partly because the term is relatively new, and everyone defines it slightly differently.

The Anthropic team made this more clear in their recent blog post, Building Effective Agents. Specifically, they make a distinction between workflows and agents. The main difference is that agents **dynamically** direct their processes; there is no fixed path.

Core components of agents

Generally speaking, three core components make up a great agent.

Memory: LLMs are stateless. They don’t "remember" anything from previous interactions unless you program them to. There are several ways to handle memory in a chat-style experience, which we covered in detail on our blog.

Aside from managing message history, most major LLM providers now support some form of prompt caching. This allows you to cache certain information, such as a long system message, so the model doesn’t need to process it from scratch with every subsequent request—reducing latency and cost.

Tools: Tools enable agents to interact with the outside world, like performing a Google Search or querying a database. When setting up tools, you should put as much effort into their configuration— name, description, etc—as you do into crafting your prompts. Think of the LLM as a developer on your team; the better you document the tool, the easier it will be to use correctly.

Planning: How you guide your agent's planning and interaction with users is crucial. Reasoning models simplify planning by leveraging built-in chain-of-thought reasoning. Still, planning—and re-planning after failed attempts—is really important for agents.

Core principles of effective prompt engineering

The basic principles of prompt engineering apply to writing prompts for agents as well.

Here's what you should keep in mind:

- **Clarity:** Simple language > complex language
- **Context:** Often we forget how much context is in our head. We need to dump all of that out and share it with the agent.
- **Specificity:** More words don't always make a better prompt. Get a teammate to read your prompt to see if they can make out the instructions.
- **Iterative Refinement:** Prompt engineering is an iterative process - no way around it! The faster you can get testing, the faster you can learn.

AI agent example 1: Cline

The first example we'll look at is the system message for Cline, a very popular and powerful coding agent in your IDE.

You can check out the full system message here. I would suggest opening it up in another tab so that you can check it out as we run through some of the cooler aspects of it.

Structured tool usage

Every action the agent takes must follow a strict XML-like syntax with explicit parameters. Setting formatting constraints and giving an example will help the agent use tools consistently and also make debugging easier for developers.

Example from the prompt:

> Tool use is formatted using XML-style tags. The tool name is enclosed in opening and closing tags, and each parameter is similarly enclosed within its own set of tags. Here's the structure:
> ```
> <tool_name>
>   <parameter1_name> value1 </parameter1_name>
>     <parameter2_name> value2 </parameter2_name>...
> </tool_name>
> ```
>
> **For example:**
> ```
> <read_file>
>     <path>src/main.js</path>
> </read_file>
> ```
> Always adhere to this format for the tool use to ensure proper parsing and execution.

Iterative, step-by-step process with confirmation

The agent can only use one tool per message and must wait for confirmation after each execution. This creates a feedback loop that minimizes the risk of cascading errors and going off in the wrong direction.

Example from the Prompt:

> It is crucial to proceed step-by-step, waiting for the user's message after each tool use before moving forward with the task. This approach allows you to:
> 1. Confirm the success of each step before proceeding.
> 2. Address any issues or errors that arise immediately.
> 3. Adapt your approach based on new information or unexpected results.
> 4. Ensure that each action builds correctly on the previous ones.

The system prompt provides a number of tools for various tasks: file operations, command executions, etc.

Each tool is clearly defined separately, and documented with instructions on when and how to use it, plus examples.

Example from the prompt (Tool descriptions):

```
##execute_command:
Description: request to execute a CLI command on the system. Use this when you need to
perform system operations or run specific commands to accomplish any step in the user's
task...
<execute_command>
    <command>Your command here</command>
    <requires_approval>true or false</requires_approval>
</execute_command>

##replace_in_file:
Description: Request to replace sections of content in an existing file using
SEARCH/REPLACE blocks that define exact changes to specific parts of the file. This tool
should be used when you need to make targeted changes to specific parts of a file.

Usage:
<replace_in_file>
<path>File path here</path>
<diff>Search and replace blocks here</diff>
</replace_in_file>
```

Plan mode vs. Act mode

The system differentiates between planning and execution.

In PLAN MODE, the agent gathers context, asks clarifying questions, and brainstorms ideas. Once a clear strategy is in place, it switches to ACT MODE to execute the plan step-by-step, minimizing errors and ensuring smooth task completion.

Example from the prompt

> ACT MODE V.S. PLAN MODEIn each user message, the environment_details will specify the current mode.
>
> There are two modes:
>   - ACT MODE: In this mode, you have access to all tools EXCEPT the plan_mode_response tool.
>   - In ACT MODE, you use tools to accomplish the user's task. Once you've completed the user's task, you use the attempt_completion tool to present the result of the task to the user.
>   - PLAN MODE: In this special mode, you have access to the plan_mode_response tool. - In PLAN MODE, the goal is to gather information and get context to create a detailed plan for accomplishing the task, which the user will review and approve before they switch you to ACT MODE to implement the solution.
>   - In PLAN MODE, when you need to converse with the user or present a plan, you should use the plan_mode_response tool to deliver your response directly, rather than using <thinking> tags to analyze when to respond. Do not talk about using plan_mode_response - just use it directly to share your thoughts and provide helpful answers.
>
> For example, if the user's task is to create a website, you may start by asking some clarifying questions, then present a detailed plan for how you will accomplish the task given the context, and perhaps engage in a back and forth to finalize the details before the user switches you to ACT MODE to implement the solution.

working directory, operating system details). Having this information reduces the risk of errors like editing incorrect files or trying to execute incompatible commands.

Example from the prompt:

> Before using the execute_command tool, you must first think about the SYSTEM INFORMATION context provided to understand the user's environment and tailor your commands to ensure they are compatible with their system.
> ====
> SYSTEM INFORMATION
> Operating System: \${osName()}
> Default Shell: \${getShell()}
> Home Directory: \${os.homedir().toPosix()}
> Current Working Directory: \${cwd.toPosix()}
> ====

Robust guidelines for editing files

Editing files is one of the most important tasks for an AI coding agent. The system prompt does a good job guiding the agent on when to use file-editing tools like write_to_file versus replace_in_file.

Example from the prompt

> EDITING FILES
>
> You have access to two tools for working with files: **write_to_file** and **replace_in_file**. Understanding their roles and selecting the right one for the job will help ensure efficient and accurate modifications.
>
> # write_to_file
> ## Purpose- Create a new file, or overwrite the entire contents of an existing file.
> ...
>
> # replace_in_file
> ## Purpose- Make targeted edits to specific parts of an existing file without overwriting the entire file.

Safety, clarity, and minimal risk

Every command or file operation must be accompanied by a clear explanation. This process of thinking before acting is similar to chain of thought prompting. Letting the model think can help it avoid making errors, or shortsighted plans.

Example from the prompt:

> Before using the execute_command tool, you must first think about the SYSTEM INFORMATION context provided to understand the user's environment and tailor your commands to ensure they are compatible with their system. You must also consider if the command you need to run should be executed in a specific directory outside of the current working directory

The TL;DR of what the Cline system message does well:

- **Structured syntax:**
  Using an XML-like format, with examples, ensures that tool use is clear and easy to debug.
- **Iterative execution:**
  Only using one tool per message and waiting for user confirmation creates a robust

Tools are laid out in a modular fashion and are well documented, helping the model select the right tool for the right task.

- **Strategic planning:**
  The clear separation between PLAN MODE and ACT MODE encourages thoughtful planning before execution.
- **Context awareness and safety:**
  Detailed system information and safety guidelines help tailor actions to the user's environment while minimizing risks.

AI agent example 2: Bolt

Next up is Bolt.new, from the company Stackblitz. Bolt is a coding agent that you can use in your browser to create and deploy apps. I'm personally a big fan and use it often.

You can check out the full system message here. I would suggest opening it up in another tab so that you can check it out as we run through some of the cooler aspects of it.

Setting constraints

The first portion of the Bolt system message is all about establishing the constraints for the model. This process of setting up the context for the model is really important.

Excerpt from the prompt:

> All code is executed in the browser. It does come with a shell that emulates zsh. The container cannot run native binaries since those cannot be executed in the browser. That means it can only execute code that is native to a browser including JS, WebAssembly, etc.
>
> • There is NO \pip\ support! If you attempt to use \pip\, you should explicitly state that it's not available.
> • CRITICAL: Third-party libraries cannot be installed or imported.
> • Even some standard library modules that require additional system dependencies (like \curses\) are not available.
> • Only modules from the core Python standard library can be used
>
> IMPORTANT: Prefer using Vite instead of implementing a custom web server.
> IMPORTANT: Git is NOT available.

Code formatting and diff specifications

Using XML, Bolt explicitly enforces code formatting instructions, such as 2-space indentation, to maintain consistency across files. Additionally, the system message provides a detailed diff specification for file modifications with a nice example.

Excerpt from the prompt:

> <code_formatting_info>
>   Use 2 spaces for code indentation
> </code_formatting_info>
> <diff_spec>
> For user-made file modifications, a <modifications> section will appear at the start of the user message. It will contain either <diff> or <file> elements for each modified file:
>   - <diff path="/some/file/path.ext">: Contains GNU unified diff format changes
>   - <file path="/some/file/path.ext">: Contains the full new content of the file

> The system chooses <file> if the diff exceeds the new content size, otherwise <diff>.

> - X: Original file starting line
> - Y: Original file line count
> - A: Modified file starting line
> - B: Modified file line count
> - (-) lines: Removed from original
> - (+) lines: Added in modified version
> - Unmarked lines: Unchanged context

```
 Example:
 <modifications>
   <diff path="/home/project/src/main.js">
     @@ -2,7 +2,10 @@
     return a + b;
     -console.log('Hello, World!');
     +console.log('Hello, Bolt!');
     +
     function greet() {
     -  return 'Greetings!';
     +  return 'Greetings!!';
     }
     +
     +console.log('The End');
   </diff>
   <file path="/home/project/package.json">
     // full file content here
   </file>
 </modifications>
</diff_spec>
```

Artifact creation and modular actions

Bolt uses artifacts to manage projects. The system message gives details about what artifacts are and how they work. Artifacts are wrapped in `<boltArtifact>` and `<boltAction>` tags, with 14 rules guiding artifact creation, modification, and communication of changes to the user. A nice example is also included as well to guide the model.

Excerpt from the prompt:

```
<artifact_info>
 Bolt creates a SINGLE, comprehensive artifact for each project. The artifact
contains all necessary steps and components, including:
 • Shell commands to run, including dependencies to install using a package
manager (NPM)
 • Files to create and their contents
 • Folders to create if necessary
</artifact_info>
<artifact_instructions>
 1. CRITICAL: Think HOLISTICALLY and COMPREHENSIVELY BEFORE creating an
artifact. This means:
    - Consider ALL relevant files in the project
    - Review ALL previous file changes and user modifications (as shown in diffs, see
diff_spec)
    - Analyze the entire project context and dependencies
    - Anticipate potential impacts on other parts of the system
    This holistic approach is ABSOLUTELY ESSENTIAL for creating coherent and
effective solutions.

 2. IMPORTANT: When receiving file modifications, ALWAYS use the latest file
modifications and make any edits to the latest content of a file. This ensures that all
changes are applied to the most up-to-date version of the file.

 3. The current working directory is ${cwd}.
```

5. Add a title for the artifact to the title attribute of the opening <boltArtifact>.

6. Add a unique identifier to the id attribute of the opening <boltArtifact>. For updates, reuse the prior identifier. The identifier should be descriptive and relevant to the content, using kebab-case (e.g., "example-code-snippet").

7. Use <boltAction> tags to define specific actions to perform.

8. For each <boltAction>, add a type to the type attribute (e.g., "shell" for shell commands or "file" for file operations). For file actions, include a filePath attribute specifying the file's relative path.

9. The order of the actions is VERY IMPORTANT. For example, create the file before running a shell command that executes it.

10. ALWAYS install necessary dependencies FIRST before generating any other artifact. If that requires a package.json, create that first.
IMPORTANT: Add all required dependencies to the package.json already and try to avoid "npm i <pkg>" if possible!

11. CRITICAL: Always provide the FULL, updated content of the artifact. This means:
- Include ALL code, even if parts are unchanged
- NEVER use placeholders like "// rest of the code remains the same..." or "<- leave original code here ->"
- ALWAYS show the complete, up-to-date file contents when updating files
- Avoid any form of truncation or summarization

12. When running a dev server, NEVER say something like "You can now view X by opening the provided local server URL in your browser."

13. If a dev server has already been started, do not re-run the dev command when new dependencies are installed or files are updated.

14. IMPORTANT: Use coding best practices and split functionality into smaller modules instead of placing everything in a single file.
- Ensure code is clean, readable, and maintainable.
- Adhere to proper naming conventions and consistent formatting.
- Split functionality into smaller, reusable modules.
- Use imports to connect these modules effectively.

ULTRA IMPORTANT: Do NOT be verbose and DO NOT explain anything unless the user asks for more information.
ULTRA IMPORTANT: Think first and reply with the artifact that contains all necessary steps to set up the project, files, and shell commands to run. It is SUPER IMPORTANT to respond with this first.
</artifact_instructions>

```
<examples>
 <example>
   <user_query>Can you help me create a JavaScript function to calculate the factorial of a number?</user_query>
   <assistant_response>
     Certainly, I can help you create a JavaScript function to calculate the factorial of a number.
     <boltArtifact id="factorial-function" title="JavaScript Factorial Function">
       <boltAction type="file" filePath="index.js">
         function factorial(n) {
           // function implementation here
         }
         console.log(factorial(5));
       </boltAction>
       <boltAction type="shell">
         node index.js
       </boltAction>
     </boltArtifact>
   </assistant_response>
```

Planning, context awareness, and order management

Before artifact creation, Bolt is instructed to do some planning and thinking ahead. It is instructed to consider the whole codebase before starting any new edits. This ensures the agent is always aware of the project environment. Combining planning with managing the order of operations in artifact creation helps Bolt deliver magical experiences.

Excerpt from the prompt:

> Think HOLISTICALLY and COMPREHENSIVELY BEFORE creating an artifact. This means:
> • Consider ALL relevant files in the project
> • Review ALL previous file changes and user modifications
> • Analyze the entire project context and dependencies
> • Anticipate potential impacts on other parts of the system.

> ”“The order of the actions is VERY IMPORTANT. … Create the file before running a command that would execute it.”

TL;DR of what the Bolt system prompt does well

- **Environment awareness:**
  Clearly defines operational constraints in WebContainer, ensuring all actions comply with the browser-based, limited environment.
- **Strict code formatting:**
  Enforces 2-space indentation and detailed diff specifications to maintain consistency and traceability in file modifications.
- **Holistic planning:**
  Emphasizes comprehensive planning—considering all files, dependencies, and context—to ensure robust and error-free solutions.
- **Modular action sequencing:**
  Uses `<boltArtifact>` and `<boltAction>` tags to structure solutions into coherent, step-by-step actions that respect order and dependency.
- **Safety and clarity:**
  Provides explicit guidelines on safe command execution and context-tailored operations to minimize risks in a constrained environment.

Conclusion

Agents are here, and they’ll only get better. The basic principles of prompt engineering apply to agents just as they do to other prompt engineering tasks.This guide will help you as you begin to work on agents, but a more valuable way to learn is the collection of prompts from teams that have already shipped agents to production.

Dan Cleary
Founder

Join thousands of AI builders

Collaborate with thousands of AI builders to discover, manage, and improve prompts—free to get started.

Start for free Schedule a demo

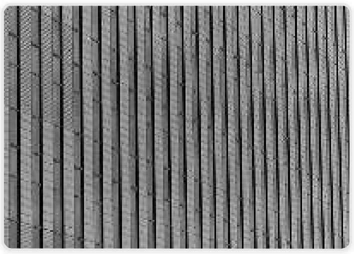
More from the PromptHub Blog

LLMs Are Eating the Context Layer

October 23, 2025

OpenAI DevDay 2025 Roundup: Apps, Agents, and the New AI Stack

October 13, 2025



Everything You Need to Know about Claude 4.5

October 2, 2025