Zed's Smart Edit Architecture: A Unified Approach to AI-Powered Code Editing

Research Paper

November 15, 2025

1 Introduction

Smart edit refers to the ability of a code editor to automatically apply changes to files when a user requests modifications through a language model interface. Unlike traditional code completion, smart edit allows users to describe desired changes in natural language, and the system intelligently determines which code to modify and safely applies edits, often across multiple files.

This paper examines how Zed, a modern code editor, implements smart edit functionality through its Agent Panel chat interface. We focus specifically on how the system handles three distinct scenarios: online language models, offline models with tool calling capabilities, and offline models without tool calling support.

2 Unified Smart Edit Pipeline

Zed's smart edit system is built around a unified architecture that handles all three model types through a single pipeline. The core components are:

- **Agent Panel**: The chat interface where users interact with language models
- **EditFileTool**: A tool that validates paths, opens buffers, and manages the editing process
- **EditAgent**: The component that renders prompts, parses model responses, and applies edits
- **Edit Parser**: Handles parsing of structured edit formats (XML tags or diff-fenced blocks)
- **Streaming Fuzzy Matcher**: Matches edit instructions against actual file content

The pipeline works as follows: when a user requests a change in the Agent Panel, the model (if tool-capable) calls the `edit_file` tool. This tool validates the file path, opens the buffer, and delegates to EditAgent. EditAgent renders a prompt template specific to the model's preferred format, sends it to the model, and streams the response back. The response is parsed into structured edits, matched against the buffer using fuzzy matching, and applied incrementally while showing a diff UI to the user.

3 Online Language Models

Online models include hosted providers such as OpenAI, Anthropic, Google AI, and GitHub Copilot. These models run in the cloud and are accessed via HTTPS with API keys.

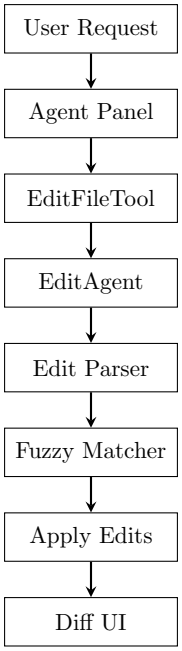

Figure 1: Unified Smart Edit Pipeline Architecture

3.1 How It Works

When using an online model with tool calling support, the flow is straightforward:

1. User types a request in the Agent Panel (e.g., "Add error handling to the login function")
2. The model receives the conversation context, including available tools
3. The model calls the `edit_file` tool with a path and description
4. EditFileTool validates the path, opens the buffer, and creates a diff view
5. EditAgent renders a prompt template (XML tags for most models, diff-fenced for Gemini)
6. The model streams back structured edits in the chosen format
7. The parser extracts `<old_text>` and `<new_text>` pairs (or diff markers)
8. The fuzzy matcher locates the old text in the buffer
9. Edits are applied incrementally and shown in a diff drawer
10. User reviews and accepts or rejects changes

The key advantage of online models is their high quality and full tool-use support. Models like Claude and GPT-4 can reason about multi-file changes, search the codebase, and apply complex edits. However, this comes with latency and cost, and data leaves the local machine.

3.2 Provider Abstraction

Zed uses a provider abstraction layer that allows the Agent Panel to swap models mid-conversation, inspect tool calls, and display token usage without vendor-specific code. This means the same smart edit flow works whether using OpenAI, Anthropic, or any other supported provider.

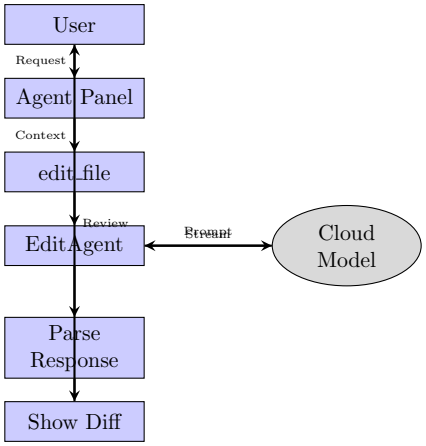

Figure 2: Online Model Flow with Tool Calling

4 Offline Models With Tool Calls

Offline models with tool calling support include self-hosted solutions like Ollama, LM Studio, or custom gateways that expose OpenAI-compatible APIs. These models run entirely on the user's machine or local network.

4.1 Architecture

The architecture for offline tool-capable models is nearly identical to online models. The main difference is that the model endpoint is local (e.g., `http://localhost:11434` for Ollama). Zed doesn't distinguish between local and remote endpoints at the protocol level.

4.2 Implementation Details

When a local model advertises `supports_tools: true`, Zed attaches tool schemas to requests. The model can then call tools like `read_file`, `list_directory`, and `edit_file` just like online models. The EditAgent uses the same prompt templates and parsing logic.

A key implementation detail is how Zed handles tool choice. When a model reports `supports_tool_choice(LanguageModelToolChoice::None)`, Zed attaches tool definitions but forces the initial call to be a normal response. This allows caching and safety checks to work even if the gateway throttles full tool streaming.

4.3 Advantages

Offline models with tool calls provide the best of both worlds: full autonomy (the model can read files, search, and edit) while keeping all data and computation local. This is ideal for privacy-sensitive environments or when working with proprietary codebases.

5 Offline Models Without Tool Calls

Some local models, particularly smaller ones (7B parameters or less), may only support plain text completion without structured tool calling. Zed handles this scenario by using a text-only editing mode.

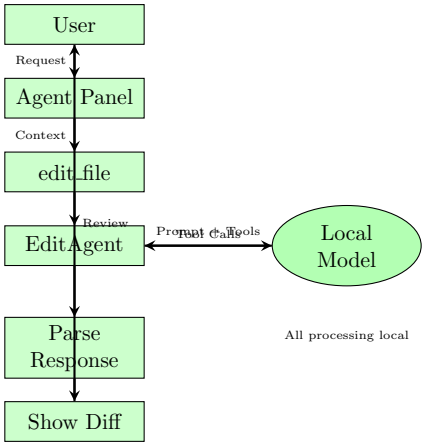

Figure 3: Offline Model Flow with Tool Calling

5.1 Text-Only Editing Mode

When a model doesn't support tools, Zed disables tool definitions and uses a simplified prompt that explicitly requests structured output. The model is instructed to produce edits in one of two formats:

- **XML tags**: `<old_text>...</old_text><new_text>...</new_text>`
- **Diff-fenced blocks**: `<<<<<<< SEARCH ... ======= ... >>>>>>> REPLACE`

The prompt explicitly states that tool calls are disabled and the response must start with `<edits>` (for XML format) or use diff markers.

5.2 Parsing and Application

The EditParser handles both formats. As the model streams its response, the parser extracts edit pairs. The StreamingFuzzyMatcher then locates the old text in the buffer, tolerating whitespace differences and partial matches.

Before committing changes, Zed validates syntax using tree-sitter, checks that ranges are valid, and confirms the old text matches within tolerance. If validation fails, the edit is rejected and an error is shown in the Agent Panel.

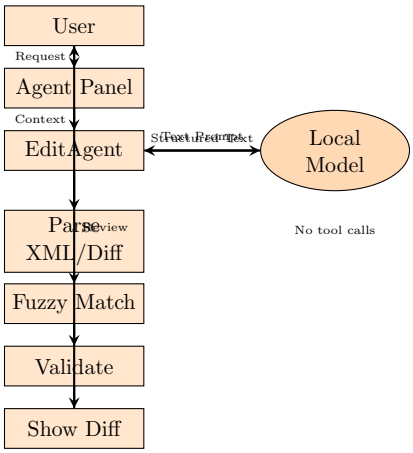

Figure 4: Offline Model Flow without Tool Calling

5.3 Limitations

Text-only mode has limitations compared to tool-calling mode:

- The model cannot open additional files mid-edit
- All context must be provided upfront in the conversation
- Formatting errors in the XML or diff blocks cause rejection
- No multi-step autonomy (the model can't search the codebase)

However, this mode is sufficient for simple, single-file edits and works with any model that can follow the structured format.

5.4 Configuration

A single setting controls this behavior: `capabilities.tools: false` in the model configuration tells Zed to skip tool definitions and use text-mode prompts. This makes it easy to migrate any offline model into the smart edit flow.

6 Edit Format Selection

Zed automatically selects the optimal edit format based on the model. Most models use XML tags (`<old_text>` and `<new_text>`), but Google's Gemini models prefer diff-fenced blocks. The EditFormat is determined by checking the model's provider ID and name.

The parser handles both formats seamlessly, extracting the same structured edit information regardless of format. This allows the same fuzzy matching and application logic to work for all models.

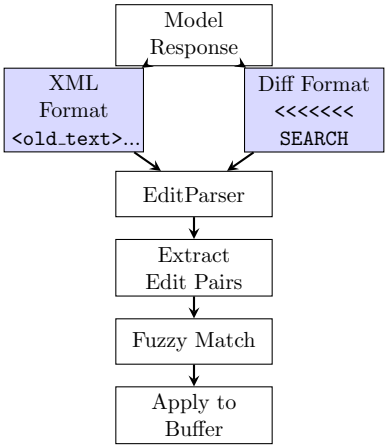

Figure 5: Edit Parsing and Application Process

7 Safety and Validation

Zed includes several safety mechanisms:

- **Path validation**: EditFileTool validates that paths are within the project before allowing edits

- **Authorization**: Edits to settings files or paths outside the project require user confirmation
- **Fuzzy matching**: The StreamingFuzzyMatcher ensures edits match actual file content, preventing hallucinations
- **Syntax validation**: Tree-sitter validates syntax before applying changes
- **Diff review**: All edits are shown in a diff view that users must explicitly accept

8 Comparison of Model Types

Figure 6 illustrates the key differences between the three model deployment scenarios. While the underlying architecture remains the same, the interaction patterns and capabilities vary based on model support.

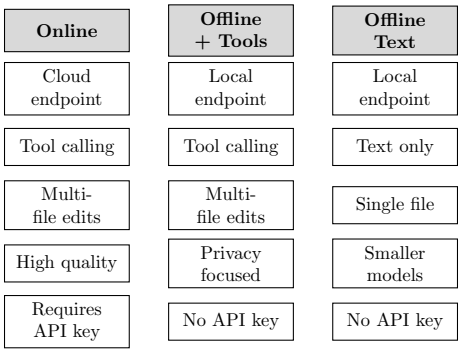

Figure 6: Comparison of Three Model Deployment Scenarios

9 Conclusion

Zed's smart edit system demonstrates a unified approach to AI-powered code editing that works across three distinct model deployment scenarios. By abstracting the differences between online and offline models, and between tool-calling and text-only models, Zed provides a consistent user experience while maximizing compatibility.

The key insight is that smart edit doesn't require tool calling—it can work with any model that can produce structured text output. This makes the feature accessible to a wide range of models, from high-end cloud services to small local models, while maintaining safety through validation and review mechanisms.

The architecture's strength lies in its flexibility: the same EditAgent, parser, and fuzzy matcher handle all three scenarios, with only the prompt format and tool availability changing based on model capabilities. This design allows Zed to provide smart edit functionality regardless of where or how the language model runs.