Zed's Context Management Architecture: A Transparent Text-First Approach to LLM Context

Research Paper

November 15, 2025

1 Introduction

Unlike many AI-powered code editors that use hidden embedding systems and vector databases, Zed takes a radically transparent, text-first approach to context management. The language model sees exactly what the user sees in the Agent Panel—there are no opaque vector stores or secret RAG layers. This paper examines how Zed builds, manages, and delivers context to language models through its text-thread architecture, explicit context injection mechanisms, and tool-based automatic context gathering.

2 The Text-First Philosophy

Zed's core principle is transparency: everything the LLM receives is visible in the Agent Panel. The context is built from explicit text blocks that users can see, edit, and control. There is no hidden embedding layer that automatically indexes the entire codebase.

- **Text Threads**: The conversation is stored as editable text blocks (System, You, Assistant)
- **Explicit Context**: All context comes from manual inclusion or tool calls
- **No Hidden Memory**: The prompt is the thread—there is no invisible history
- **Editable History**: Users can rewrite any part of the conversation before resending

This approach gives users complete control and visibility over what context the model receives, making it easier to debug, optimize, and understand model behavior.

3 Thread-Based Context Architecture

Zed's context management centers around the concept of a "text thread"—a specialized document that contains the entire conversation history.

When a user sends a message, Zed converts these text blocks into a standard chat messages array (system, user, assistant) and sends it to the chosen provider. The thread itself is the prompt—there is no separate "memory" object.

4 Building Context: Manual Injection Methods

Zed provides several explicit ways to inject context into a thread. All of these methods result in literal text being inserted into the conversation.

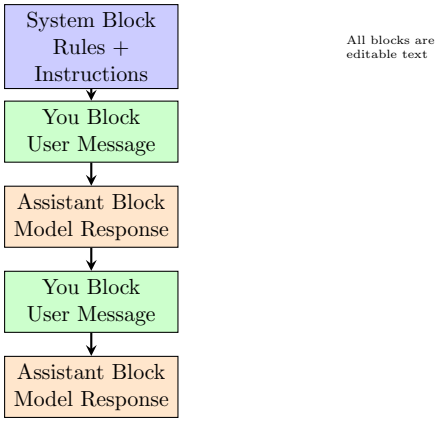

Figure 1: Text Thread Structure

4.1 Slash Commands

Slash commands are the primary mechanism for adding context. When executed, they resolve once and paste their results directly into the thread as text.

- `/file <path>` – Inserts file contents or directory tree
- `/tab [name|all]` – Inserts current tab or all open tabs
- `/selection` – Inserts the current text selection
- `/diagnostics` – Inserts language-server diagnostics
- `/terminal [n]` – Inserts last N lines of terminal output
- `/now` – Inserts current date/time
- `/symbols` – Inserts symbol outline of current file
- `/prompt <name>` – Inserts saved rules
- `/fetch <url>` – Fetches content from HTTP endpoint

Important: Folds in the UI are just visual—the model still sees all the text. Token counts include folded content.

4.2 @-Mentions

The Agent Panel supports @-mentions for adding context:

- @files – Mention specific files
- @directories – Mention entire directories
- @symbols – Mention code symbols
- @threads – Reference previous conversations
- @rules – Reference saved rule files

Mentions are resolved when building the final request, similar to slash commands. They expand into tagged sections within a `<context>` block, clearly labeled so the LLM knows what was attached by the user.

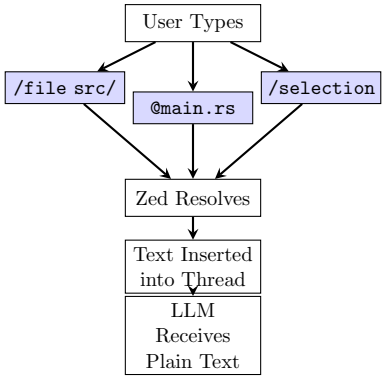

Figure 2: Context Injection Flow

5 System Prompt Construction

The system prompt is built from multiple sources and is always prepended to every request.

5.1 Rules and Project Context

Zed automatically scans project directories for rule files matching patterns like:

- `.rules`
- `.cursorrules`
- `.windsurfrules`
- `.clinerules`
- `.github/copilot-instructions.md`
- `AGENT*.md`, `CLAUDE.md`, `GEMINI.md`

These rules are loaded and injected into the system prompt. Additionally, the system prompt includes:

- Fixed instructions (communication style, tool usage rules)
- List of visible worktree roots
- Operating system, architecture, and shell information
- Model name and capabilities
- User's custom rules from the Rules Library

6 Automatic Context via Tools

While manual context injection gives users control, Zed's built-in agent can also automatically gather context through tool calls.

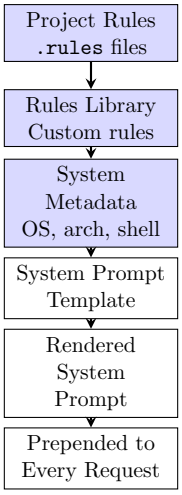

Figure 3: System Prompt Construction

6.1 Built-in Tools

Zed's agent has access to tools that allow it to:

- Read files from the project
- Search the codebase for symbols
- Run terminal commands
- Edit files (as discussed in the smart edit paper)
- List directories

When the agent calls a tool, the tool's output is added as a context message in the conversation. The agent can see these results and use them to generate its response.

6.2 Profiles

Zed provides different profiles (Write, Ask, Minimal) that control which tools are available:

- **Write**: Full tool access including file editing
- **Ask**: Read-only tools (read files, search, but no edits)
- **Minimal**: No tools, just conversation

Users can see which tools are being used during a response in the Agent Panel UI.

7 External Context via MCP

The Model Context Protocol (MCP) allows Zed to integrate with external context servers that provide additional data sources.

7.1 How MCP Works

MCP servers expose external data such as:

- Database schemas (Postgres, PlanetScale)

- GitHub repositories and issues
- Web search results (Exa Search)
- Analytics and telemetry data

When a user invokes an MCP command:

1. Zed sends a request to the MCP server
2. The server runs its logic (which may include embeddings/RAG internally)
3. The server returns text
4. Zed pastes that text into the thread

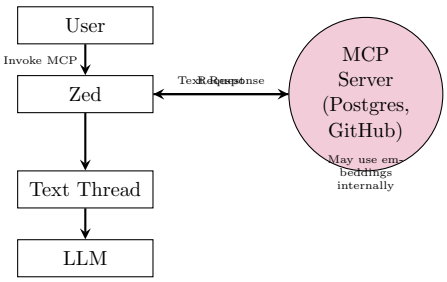

Figure 4: MCP Context Integration

Key point: Zed itself doesn't manage embeddings. Any vector search happens inside the MCP server, and Zed only receives the final text result.

8 Session Memory and Summarization

8.1 Thread Persistence

Threads are stored persistently:

- All messages (user, assistant, tool calls) are saved
- Threads can be reopened from history
- Previous threads can be @-mentioned to pull their content into new conversations

Storage format: Threads are serialized as JSON or zstd-compressed JSON in `threads/threads.db`. There is no vector database or embedding index.

8.2 Token Pressure and Summarization

Zed tracks token usage and shows a meter in the UI. When approaching the model's context limit:

1. Zed suggests starting a new thread
2. The current thread can be summarized using a separate LLM call
3. The summary becomes the starting context for the new thread
4. From that point, the summary text is what persists as "memory"

This is rolling textual summarization, not vector-based retrieval. Long-term memory is implemented as text summaries that users can see and edit.

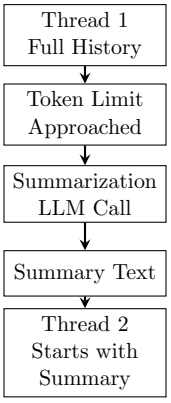

Figure 5: Thread Summarization Flow

9 Complete Context Flow

Putting it all together, here's how context flows from user input to the LLM:

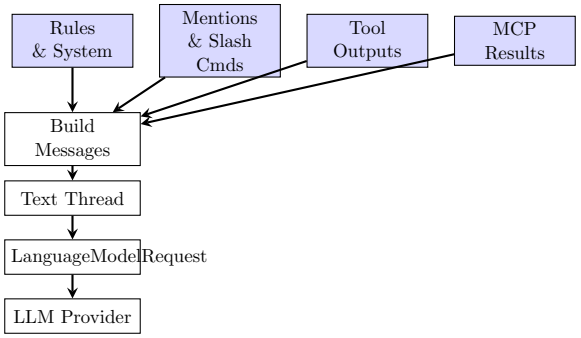

Figure 6: Complete Context Assembly Pipeline

The final request contains:

1. **System messages**: Built from rules, project context, and system metadata
2. **User messages**: Current input plus any retained history
3. **Assistant messages**: Previous model responses (if kept)
4. **Context blocks**: Resolved mentions, slash commands, tool outputs, MCP results
5. **Tool schemas**: Available tools and their definitions

Everything is visible in the Agent Panel—there are no hidden context layers.

10 Online vs Offline: Context Consistency

The context building process is identical regardless of whether the model is online or offline:

- **Online (hosted)**: Same text-thread pipeline, full tool support
- **Offline (Ollama/LM Studio)**: Same text-thread pipeline, tools work if model supports them
- **Offline (no tools)**: Same text-thread pipeline, but only explicit context (no automatic tool calls)

The difference is in transport and tool capabilities, not in how context is built or stored.

11 What Zed Does Not Have

It's important to understand what Zed explicitly does not include:

- **No built-in vector store**: Core Zed does not maintain an embedding index of the codebase
- **No automatic RAG**: There's no hidden retrieval-augmented generation layer
- **No opaque memory**: Everything the LLM sees is visible in the thread
- **No secret context**: No background processes that silently add context

If you need vector search or semantic retrieval, you must:

- Use an MCP server that implements it (and returns text to Zed)
- Build a custom Agent Server that does embeddings internally
- Implement it in an external service that Zed calls via MCP

12 Advantages of the Text-First Approach

Zed's transparent, text-first context management offers several benefits:

- **Debuggability**: Users can see exactly what context the model received
- **Control**: Users decide what context to include, when, and how
- **Predictability**: No hidden behavior that's hard to reason about
- **Editability**: History can be rewritten before resending
- **Composability**: Context sources (rules, files, tools, MCP) are clearly separated
- **Token transparency**: Users can see token usage and optimize accordingly

13 Conclusion

Zed's context management architecture demonstrates that sophisticated AI assistance doesn't require hidden embedding systems or opaque vector stores. By making context explicit, editable, and visible, Zed gives users the control and transparency needed to effectively work with language models.

The text-first approach means:

- Context is built from explicit text inclusion
- Everything is visible in the Agent Panel
- Memory is stored as editable text threads
- Long-term context uses summarization, not vectors
- External embeddings (via MCP) are opt-in and transparent

This architecture provides a solid foundation for building reliable, debuggable AI-powered development tools while maintaining user control and system transparency.