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

Research Paper

November 17, 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 or Text Threads—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 observable automatic context gathering via tools.

## 2 The Text-First Philosophy

Zed's core principle is transparency: everything the LLM receives is visible in the Agent Panel or Text Threads. 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 observable 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
- **Automatic but Observable**: Agent-driven context gathering is visible, not hidden

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 two distinct thread types, each serving different purposes.

### 3.1 Text Threads vs Agent Panel

Zed provides two thread interfaces with different capabilities:

- **Text Threads**: Editor-like, fully editable conversation documents. Support slash commands but have no tool calls or agentic behavior—they're "just chat" for simple Q&A and file inspection workflows.

- **Agent Panel Threads**: Optimized for agentic workflows with @-mentions, tool calls, profiles, MCP integration, external agents, token meters, and summarization. This is where automatic context gathering and code editing happen.

Both use the same underlying text-thread architecture (editable System/You/Assistant blocks), but Agent Panel threads add the tool-calling layer that enables autonomous codebase traversal.

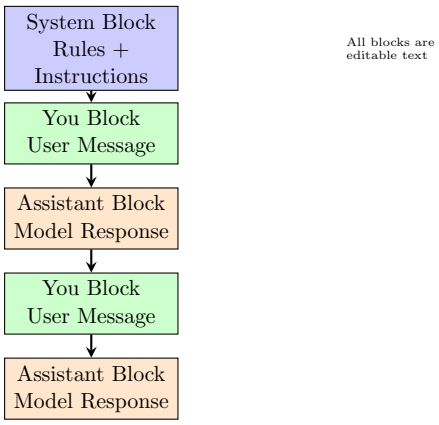


Figure 1: Text Thread Structure

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 explicit ways to inject context into threads. All methods result in literal text being inserted into the conversation.

## 4.1 @-Mentions (Agent Panel Primary)

In the Agent Panel, @-mentions are the primary mechanism for adding context:

- @files – Mention specific files
- @directories – Mention entire directories
- @symbols – Mention code symbols
- @threads – Reference previous conversations
- @rules – Reference saved rule files
- Selection – Add current selection via @ menu

Mentions expand into tagged `<context>` blocks when building the request, clearly labeled so the LLM knows what was attached.

### 4.2 Slash Commands (Text Threads & Extensions)

Slash commands are especially important in Text Threads and can be extended by extensions or external agents:

- `/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

Extensions can provide custom slash commands (e.g., MCP servers, external agents like Claude Code). Slash commands are supported in rules files when used with Text Threads, but @-mentions are not supported inside rules files.

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

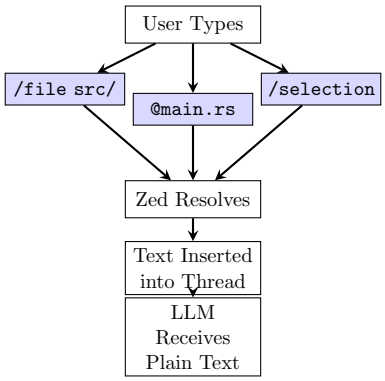


Figure 2: Context Injection Flow

## 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`

### 5.1.1 Rule Precedence and Limitations

Zed uses only the first matching project rules file in a priority list (first match wins). Rules are project-level only—there's no built-in per-directory or per-file granularity. Rules Library entries don't apply automatically; you must explicitly @-mention them (`@rules`) in the Agent Panel.

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

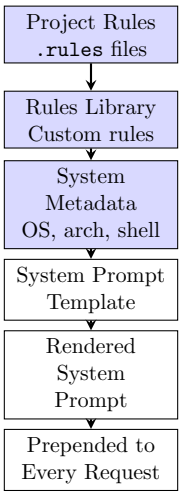


Figure 3: System Prompt Construction

## 6 Automatic Context via Tools

While manual context injection gives users control, Zed's built-in agent can automatically gather context through tool calls. This is automatic but observable: the agent chooses which tools to call and which files to read, but all tool calls and their outputs appear as visible messages in the thread.

### 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. This automatic traversal is transparent—unlike hidden embedding systems, you see every file read and every tool call.

### 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: MCP and External Agents

Zed integrates with external systems through two protocols: MCP (for data into Zed) and ACP (for routing context to external agents).

### 7.1 Model Context Protocol (MCP)

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

### 7.2 External Agents via ACP

Zed supports external agents (Claude Code, Gemini CLI, Codex) via the Agent Client Protocol (ACP). When using external agents:

- Zed still provides context via @-mentions and files, but forwards this to the external process via ACP
- Some features may be limited: editing past messages, resuming history, checkpointing
- Slash commands may behave differently or be partially supported
- The same text-first, explicit context is repacked and forwarded over ACP

This means "online models" can be either Zed-hosted (via Zed's agent) or external CLI tools (via ACP) with their own context handling.

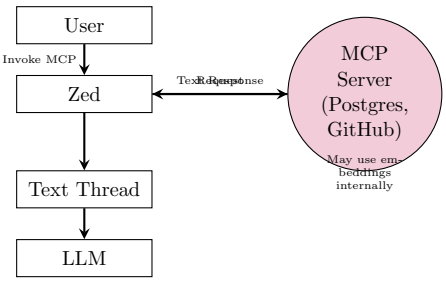


Figure 4: MCP Context Integration

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

Zed persists threads on disk (e.g., in a local database) so they can be reopened from history. The exact storage format is an implementation detail of the open-source codebase. 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. Different models have very different context windows (e.g., 128k vs 32k), which affects how aggressive users must be about summarization. Zed recommends using purpose-based Agent threads (new thread per task) as a context-management strategy.

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"

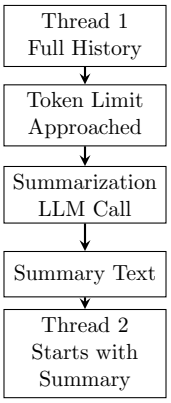


Figure 5: Thread Summarization Flow

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

## 9 Complete Context Flow

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

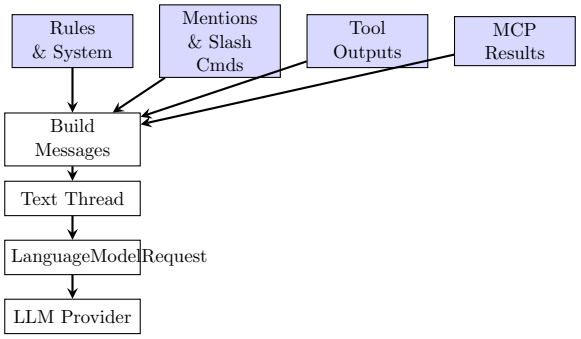


Figure 6: Complete Context Assembly Pipeline

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)
- **External agents (ACP)**: Context forwarded via ACP, may have different capabilities

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

## 11 Security and Privacy of Context

The text-first approach makes it easier to audit what data leaves your machine:

- **What's sent**: Everything in the thread, attached files, rules, and tool outputs included in the request
- **Hosted models**: Data sent to OpenAI, Anthropic, etc. (their privacy policies apply)
- **Self-hosted**: Data stays local (Ollama, LM Studio)
- **MCP servers**: Run as separate processes and may log/store what they receive—they're not automatically trusted
- **External agents**: Forward context via ACP; their privacy policies apply

Since all context is visible in the thread, users can review exactly what will be sent before making a request.

## 12 Text-First vs RAG-First: Tradeoffs

Zed explicitly does not include built-in vector stores or automatic RAG. This design choice has clear tradeoffs:

### 12.1 Advantages

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

### 12.2 Costs

- **Manual curation**: Users (or tools) must curate context—easier to miss important files in huge monorepos
- **Token usage**: Can explode if people @directory everything without filtering
- **No automatic semantic search**: Must bring MCP/external RAG for semantic retrieval across millions of lines

### 12.3 Hybrid Approaches

If you need vector search or semantic retrieval:

- Use an MCP server that implements it (returns compact textual digest)
- Build a custom Agent Server that does embeddings internally
- Use BMAD-style documentation methods that convert semantic structure into textual knowledge

This keeps RAG opt-in and transparent, rather than hidden in the core system.

## 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 (manual or tool-driven)
- Everything is visible in the Agent Panel or Text Threads
- Memory is stored as editable text threads
- Long-term context uses summarization, not vectors
- External embeddings (via MCP) are opt-in and transparent
- Automatic context gathering is observable, not hidden

This architecture provides a solid foundation for building reliable, debuggable AI-powered development tools while maintaining user control and system transparency. The tradeoff is that users must be more deliberate about context curation, but gain complete visibility and control over what the model sees.