# A Language for Describing Agentic LLM Contexts

Noga Peleg Pelc
Dept. of Computer Science and AI
Bar-Ilan University
pelegpelcnoga@gmail

Gal A. Kaminka
Dept. of Computer Science and AI
Bar-Ilan University
galk@cs.biu.ac.il

Yoav Goldberg
Dept. of Computer Science and AI
Bar-Ilan University & Ai2
yoav.goldberg@gmail

## Abstract

Large language models are increasingly used within larger systems ("LLM agents"). These make a sequence of LLM calls, each call providing the LLM with a combination of instructions, observations, and interaction history. The design of the encoded information and its structure play a central role in the quality of the resulting system, leading to efforts spent on context engineering. It is therefore critical to communicate the composition of the LLM context in a system, and how it evolves over time. Yet, no standard exists for doing so: context construction is typically conveyed through informal prose, ad hoc diagrams, or direct inspection of code, none of which precisely capture how a prompt evolves across interaction steps or how two context representation strategies differ. To remedy this, we introduce the Agentic Context Description Language (ACDL), a language for specifying the structure and dynamics of LLM input contexts in a precise, readable, and standard manner, along with visualizations. ACDL provides constructs for specifying context aspects such as role message sequences, dynamic content, time-indexed references, and conditional or iterative structure, capturing the full architecture of a prompt independently of any particular implementation. ACDL diagrams can be hand drawn on a whiteboard, or written in formal language which can then be rendered. We describe the language, demonstrate it by documenting several existing systems and their variants, and encourage the community to adopt it for describing LLM systems context, both in day-to-day communication and in papers. Tooling, examples and documentation are available at www.acdlang.org.

**ACM Reference Format:**
Noga Peleg Pelc, Gal A. Kaminka, and Yoav Goldberg. 2026. A Language for Describing Agentic LLM Contexts. In *ACM Conference on AI and Agentic Systems (CAIS '26), May 26–29, 2026, San Jose, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3786335.3813126

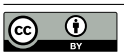
## 1 Introduction

Large language models are increasingly serving as the core reasoning component in complex, multi-component systems. Modern agentic systems, ranging from autonomous coders to multi-step reasoning frameworks, rely on sophisticated input contexts that evolve dynamically over time. The evolving structure of these contexts, including which messages are included, how history is accumulated, and under what conditions content appears, is a central design artifact that directly shapes agent behavior.

Despite the central role of context structure, there is currently no standard way to describe it, leaving practitioners to rely on informal prose, ad hoc diagrams, or direct inspection of implementation code. At the community level, the absence of a shared descriptive framework makes it challenging to compare context construction strategies across systems, especially when they appear similar at a high level. In addition, although many published systems provide accompanying code and prompt templates, the logic governing context assembly is rarely made explicit in the paper itself, complicating efforts to reproduce or meaningfully re-implement proposed methods. Finally, Within research and engineering teams, it becomes difficult to rigorously communicate how the context of an agent evolves over time or to reason about the impact of incremental changes in that context.

To address these challenges, we present **Agentic Context Description Language** (ACDL), a formal language for precisely describing the structure and evolution of context in multi-turn interactions between agents and large language models. ACDL abstracts away concrete content, replacing it with symbolic labels that describe the *role*, *type*, and *source* of each element—distinguishing, for example, system instructions from user queries, or tool outputs from model reasoning. Temporal indices track how elements accumulate across interaction steps, while control flow constructs such as loops and marked regions capture the patterns by which context is assembled and transformed. This abstraction enables crisp communication and rigorous analysis of context structure and dynamics, independently of any particular implementation.

The value of ACDL is illustrated in Figure 1, which presents three structurally distinct implementations of a multi-step ReAct[1] [10] agent rendered from their ACDL specification. All three implementations realize the same high-level design: a system prompt, user query, and iterative tool-use loop. However, they differ in where instructions appear, how assistant reasoning is interleaved with tool responses, and whether certain elements are repeated at each iteration. These variations reflect valid design choices that affect agent behavior, yet describing them in text does not easily reveal the differences between them (as we discuss in Section 5, even such small structural variations between ReAct implementations can lead to measurable differences in agent performance). ACDL renders each structural difference explicit, and the renderings are immediately comparable as differences in block locations and message roles are visible at a glance. Without a formal representation, articulating these distinctions, let alone systematically, would require lengthy prose or direct inspection of the code.




*CAIS '26, San Jose, CA, USA*

ACM ISBN 979-8-4007-2415-2/2026/05
https://doi.org/10.1145/3786335.3813126

[1] ReAct (Reasoning + Acting) is a prompting framework where a language model interleaves chain-of-thought reasoning with actions (such as calling tools or APIs) and observations from those actions, looping through thought, action, and observation steps until it reaches an answer.

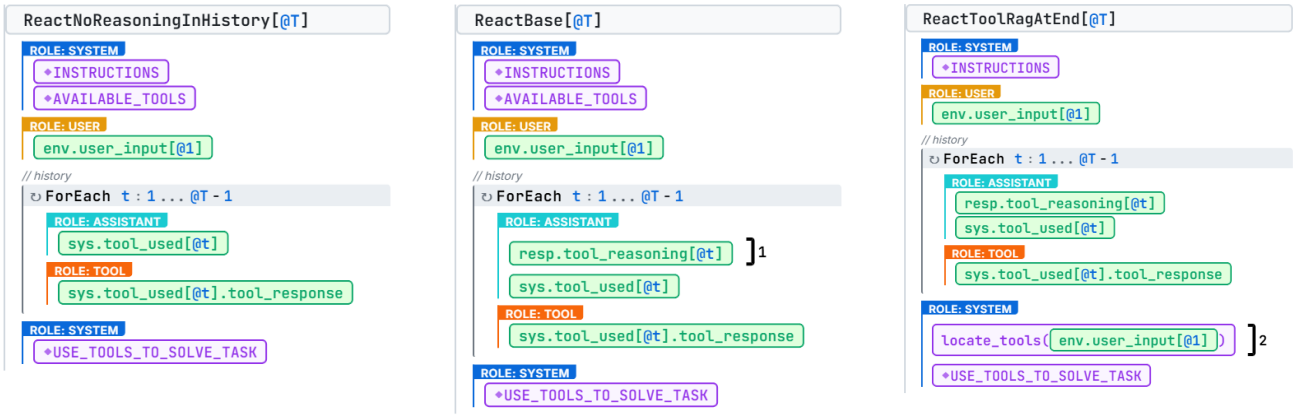



**Figure 1: ACDL visualization of three simple ReAct loop variants. Middle: base implementation. Left: no reasoning traces in action history (without [1]). Right: query-based tool selection, tools appear at last rather than first message ([2]). ACDL makes expressing and discussing such differences concise, precise, and immediately apparent.**

ACDL was designed to provide a shared language for describing and communicating about context structure in agentic LLM systems, to enable clearer communication within teams, more faithful comparison between published systems, and more transparent documentation of prompt logic in research articles and white-papers. It also has the potential for being a clear *human-to-AI* communication medium, describing the desired context layout to coding agents in unambiguous terms. In addition to the language specification, we provide a parser, an interactive renderer, a vscode plugin, and an agentic skill that operationalize ACDL.

## 2 Preliminaries: Agentic Systems Terminology

We begin by clarifying the conceptual foundations underlying agentic systems, and the core challenges involved in building them. An *agentic system* is an autonomous entity that operates in an iterative cycle of observing its environment, reasoning and making decisions about the appropriate course of action pursuing its own tasks, and taking actions according to these decisions. The decision-making process is sequential: later decisions take into account the current state as well as previous decisions and their results. An agentic system operates its own time steps, maintaining its own memory which it uses to store and later retrieve information, to inform future behavior. A multi-agent system comprises multiple agentic systems operating within a shared environment. Agentic systems within a multi-agent system may communicate with or observe the outputs of other agentic systems, and, in some instances, may share memory between agents. Together, these properties define the abstractions that the remainder of this paper builds upon.

**LLM-based Agentic Systems.** In an *LLM-based agentic system* some or all the decisions are delegated to an LLM. At each decision point, the system specifies a situation, a set of possible parameterized actions to be taken, and supporting information. This information is then packed into a prompt and sent to the LLM (potentially together with additional instructions). The information specified in the prompt is called *the context* (see *What's in a Context?* below). The system interprets the LLM's response and acts according to it. As LLM calls are independent from each other (stateless) yet the agent's decision making is ideally sequential (statefull), the context must contain enough information about the system's state and process history for the LLM to make the correct responses.

**What's in a Context?** Given the role of the context to communicate stateful information from one LLM invocation to the next, the context often includes *a history* component, listing the past turns of the interaction and their outcomes. In addition, the agentic system may provide the LLM with general purpose instructions, information about the state of the environment, information about results of actions, information retrieved from long term memory and a set of possible actions to take (or tools to call). It may also list internal state information such as current goals and subgoals, plans, progress tracking towards achieving each of the subgoals or plans, reminders, etc. Just as history accumulates and therefore changes from one invocation to the next, so do other information components included in the context.

Changes may occur because of the system's actions that affect the environment, system actions that affect internal states, or changes to the environment that are independent of the system. It is convenient to talk about "steps" that the system takes, and talk about the context sent to the LLM at a specific step $t$. All the state values discussed above are also indexed according to these steps.

**Describing Contexts vs Prompts.** We distinguish *context engineering* from the closely-related, yet distinct, *prompt engineering*. At some technical level, the terms *context* and *prompt* are interchangeable, and describe the entirety of information sent as input to the LLM in a single call. However, the terms are also used to distinguish different engineering concerns.

We use *context engineering* to explicitly and purposefully abstract away from specific word choices and similar details that are left for *prompt engineering*. Specifically, we use *prompt engineering* to refer to tuning the specific wording of instructions, persona adoption, and phrasing used to nudge an LLM toward a desired output. In

contrast, we use *context engineering* when discussing the macro-logic of information selection and structuring: what information is presented to the LLM at each stage, how it is packed into role messages, and where it is situated within the input. Context engineering also addresses labeling of tool observations, how historical turns are prioritized, and how system instructions are protected from attention dilution.

The details of the context composition have a significant impact on the LLM system's performance. The following example illustrates how such a description may be carried out today.

**Example Descriptions of LLM-Based Agentic Systems.** A simple example of an LLM-based agentic system is a chat loop[2] where at each turn of the conversation the controller presents the LLM with all previous conversation turns (or a summary of them) followed by the latest user query. Another example is a ReAct[10] tool-calling loop, in which the system presents the LLM with a set of possible actions ("tools"), asks it to reason and then select the best one, applies the action, and repeats the process until a DONE action is selected. Here, the iteration is over action-selection steps, and at each step the LLM is presented with a summary of previous tool invocations and their outcomes. A chat-bot and ReAct tool-calling scenarios can be combined into a system that converses with a user and, upon each user turn, runs an internal ReAct loop for producing a response, which is then sent back to the user. Here, the history sent to the LLM at each turn will comprise of the previous user requests, the previous system's responses to these requests, and some or all of the intermediary tool-calling requests and responses that resulted in these system responses.

The language used in the description above has been kept quite loose: it is not clear exactly how the history is structured and what information it contains or does not contain. While many papers denote more attention to refining such descriptions and making them clearer, it is in fact quite challenging. It is often the case that despite the best attempts of researchers and practitioners to clearly describe contexts in text or code, the description remains subject to interpretation. This is the motivation for the our work.

## 3 The Need for a Context Description Language

Clear communication of design choices is a prerequisite for reproducible and improvable engineering. In computer science and software engineering, commonly used visual languages expressing system design include UML class diagrams, flowcharts, sequence diagrams, state diagrams, and reactive streams marble diagrams, to name a few, with software like Mermaid [8] allowing to express such diagrams in text-based markup.

In Machine Learning, *plate-notation* expressing bayesian graphical models and *factor-graph* notations were once ubiquitous and a major driving force for research advances in these fields. In deep-learning, the lack of a standard diagramming convention has been identified as a barrier to faithful implementation, comparison, and analysis of neural network architectures [1, 5]. The now-ubiquitous box diagrams used to describe architectures like ResNet or the Transformer—while informal and ad-hoc—have become an indispensable communication tool, enabling researchers to convey structural decisions and differences at a glance. Their value lies not in executability, but in making structure visible and comparable.

A well designed domain specific language (DSL) distills the core elements of the domain it attempts to communicate and their compositional patterns, and assigns them to a clear and consistent vocabulary. By so doing, it provides a shared vocabulary that enables practitioners to describe, compare, and reason about designs at the appropriate level of abstraction. Moreover, the existence of a language for describing artifacts has the potential of elevating these artifacts from after-thoughts to rigorous objects of study. Once a language exists to document structural differences, the variability space becomes apparent, and begs investigation.

In our view, context construction in agentic LLM systems is in dire need for such a language. There is currently no standard way to precisely describe how the input to a language model is assembled and how it evolves across interaction steps. For example, in a recent tech report, DeepSeek-AI [3] describes how training data for chat conversations and training data for agentic tool use differ in the histories presented to the model. The verbal description is accompanied by two ad-hoc diagrams explaining the differences, and, while overall effective, both the text and diagrams do not fully capture the context structure. ACDL can be used to provide a unified and unambiguous description. Their text, diagrams and corresponding ACDL interpretations appear in Appendix C.

While several languages have been proposed for structuring prompts or orchestrating LLM pipelines—including PromptML [11], PDL [2], and POML [13]—these focus on organizing the content of individual prompts or on executing LLM-powered workflows, rather than on describing the mapping of temporally evolving system state and history to LLM contexts across multi-turn interactions.

As we discuss in Section 5, even small variations in how a very basic ReAct agent's context is assembled—namely the inclusion or exclusion of reasoning steps from the history—lead to measurable differences in agent performance. These are precisely the kinds of distinctions that a descriptive language should make explicit. ACDL addresses this gap: it is not a programming language for building agents, but a descriptive language for specifying how context is composed, how it changes over time, and where its components originate.

## 4 The Agentic Context Description Language

The Agentic Context Description Language (ACDL) is a a domain specific language for describing how LLM contexts are constructed as the state of the system evolves over time. An ACDL specification describes the structure of the prompt presented to the model at each interaction step: which messages appear, in what order, with what content, and under what conditions. The language is descriptive—it specifies what the context contains, not how it is computed—and is independent of any particular codebase or runtime framework. The language captures the structural blueprint of a prompt rather than its exact wording. It allows the description to refer to structural elements within past contexts or responses, and thus to describe how the context structure evolves.

[2]We take a broad view of agentic systems, which also includes chat-bots.

**Scope.** ACDL's scope is focused by design: precise description of the context window as seen by the language model (Figure 2). The context window sent to the LLM at time $T$ is derived from the state and the history at time $T$, and how they are translated to LLM contexts. ACDL describes this derivation logic. It is *not intended* to describe how state and history were produced, nor other aspects of the system: agentic logic, tool implementations, retrieval mechanisms, model behavior and so on. These are explicitly out of scope for ACDL, and are meant to be described by other means.

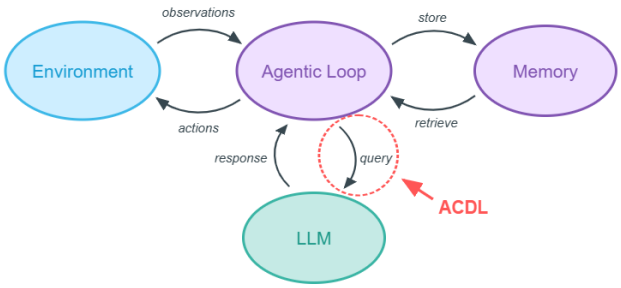



**Figure 2: ACDL is concerned with describing the queries (contexts) sent to the LLM by an agentic controller.**

ACDL is not meant to describe how state changes across steps, only how existing state and history are. are linearized into a context. A detailed description of ACDL syntax appears in Appendix D. Below, we use short examples to briefly describe major ACDL elements.

**Context as sequence of information pieces** At the most abstract description, an LLM context is a linear sequence of *information pieces*. In current LLM APIs, this sequence is packed into *role messages* where each message is labeled with one of (currently) four distinct roles (*user* (U), *assistant* (A), *system* (S), *tool* (T)), and each message contains zero or more pieces of information. Thus, an ACDL context is a sequence of *role messages* and each messages contains a sequence of information pieces.[3] Older LLM APIs (text completion) can be thought of as having a single role message with a role of *none* (N).

The following is a minimal context described in ACDL, showing a simple sequence of two information pieces. The ACDL source is on the left; the visual rendition is on the right. The ACDL code defines a context named `Context`, applied at time `@T`. We will discuss time in more detail later. For now, it suffices to say that we assume the system operates in discrete time steps, counting integers from 1 onwards, and `@T` represents the current step. Time-step variables are marked with a @ prefix.

```
Context[@T]: {
  S: PERSONA
  S: INSTRUCTIONS
}
```

```
Context[@T]
ROLE: SYSTEM
*PERSONA
ROLE: SYSTEM
*INSTRUCTIONS
```

[3]Some new APIs break this assumption by supplementing the `messages` field with additional fields such as `tools`, whose content are then linearized into the context by the LLM provider, without user control on placement or formatting. Since these fields have relatively rigid structures, we currently leave them outside of ACDL, and expect their content to be specified in a comment or an accompanying description. Extending ACDL to support such fields is straightforward, and we may do so in the future.

The Context contains a sequence of two information pieces. A single role message (S, indicating a *system* role) with the content `PERSONA`, and a second *system* message with the content `INSTRUCTIONS`. Upper case symbols like `PERSONA` and `INSTRUCTIONS` represent constant strings (templates) that are defined elsewhere. While the context is indexed by time @T, these strings do not vary with time — they are identical for all values of @T. ACDL purposefully abstracts over the exact strings (which are a matter of prompt engineering), and focuses instead on the information pieces conveyed by these strings.

**Information sources.** While the pieces of information can be arbitrary strings, it is helpful to categorize them based on their source. These include: (a) constant strings or templates; (b) values derived from the current system state; (c) values derived from the environment state; (d) values derived from previous LLM responses; and (e) values derived from functions applied to the above.

We now show how these elements manifest in ACDL. One way is using templates that take arguments, which are pieces of information to be embedded in the string. In the following example, the `INSTRUCTIONS` template takes two such arguments: a configuration-based "role" ("*You are {{an expert coder}}*") and the time at which the first step took place.

```
Context[@T]: {
  S: INSTRUCTIONS(sys.conf.role, sys.time[@1])
  U: {
    sys.time[@T]
    env.user_input[@T]
  }
}
```

```
Context[@T]
ROLE: SYSTEM
*INSTRUCTIONS( sys.conf.role , sys.time[@1] )
ROLE: USER
sys.time[@T]
env.user_input[@T]
```

It also adds a *user* message with two pieces of information: the *time* and the *user input* at *the current step*. At each time step of the system, the LLM is presented with a fixed set of instructions in a system message, followed by the latest user input, timestamped with the time in which it occurred. The user message likely contains some characters connecting the time and the user input. As ACDL focuses on the varying information and its source (rather than the exact strings), these are assumed to exist, but are not specified in ACDL.

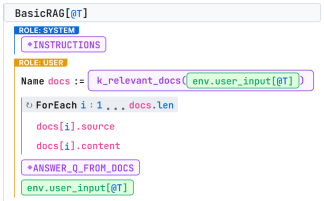
The *sys* and *env* prefixes denote global objects that hold state information: *sys* captures state maintained by the system (configuration, memory, action and tool-use history, etc.), while *env* captures external state observed from the environment (user inputs, observed world state, etc.). Although both could share a single namespace, we keep them distinct because the distinciton between internal and external state is important for reasoning about system behavior and relates to issues such as trust, mutability, and persistence. An additional prefix is *resp*, indicating an LLM response from a previous turn, whose value is used as is.

Every piece of information in the context is either constant, or derives from a state at a given time step. ACDL does not define how the state is structured, and assumes the states and their history are maintained externally. For example, accessing a property of *sys*, *env* or *resp* assumes such a property is tracked by the system and contains the specified information. Crucially, all values are immutable. A value such as *sys.conf.role* will remain the same throughout the

lifetime of the system. Values that change between states are accessed by indexing them with a state index: *env.user_input[@T]* is the value of the user input at the current step (@T).

The final kind of information source is *functions*, demonstrated here in a simple RAG example (this example also shows assigning names to expressions with the `Name` keyword, and iteration over ranges and collections with the `ForEach` keyword, described later).

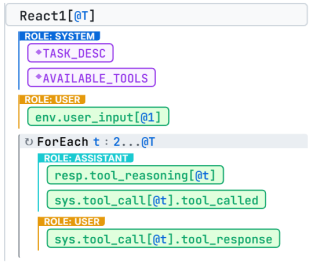
```
BasicRAG[@T]: {
    S: INSTRUCTIONS
    U: {
        Name docs := k_relevant_docs(env.user_input[@T])
        ForEach(i: range(1, $docs.len)) {
          $docs[i].source
          $docs[i].content
        }
        ANSWER_Q_FROM_DOCS
        env.user_input[@T]
    }
}
```




`k_relevant_docs` is a function, computing values based on the parameters passed to it. Functions have descriptive names, and their semantics are either inferred by the reader or specified elsewhere (not in ACDL). Here, the function returns a list of *k* documents that are based on the current user's input. Each document contains (at least) a source and a content, which are included in the context.

Functions in ACDL are assumed to be pure, and their return values depend only on their parameters and external constants: calls to the same function with the same arguments always return the same results. If the function depends on state that changes through time (e.g., a memory that evolves as the system progresses) it should be reflected in their inputs: *k_relevant_docs(env.user_input[@T], sys.mem[@T], env.DB[@T])*.

**Control Structures.** In addition to iteration (see below), ACDL also supports conditions (`If`, `Else`, `Switch`). Control structures can appear either within a role message, as in the example above, or around role messages, as in the example below (a basic ReAct tool-calling agent):




This examples demonstrates the most common use of ACDL `ForEach` iterations: unpacking history into the context. `React1` describes the context of a system that takes a user question as input, and then attempts to provide an answer by performing a series of tool calls, until it has sufficient information to answer the question reliably.

In the first time step, @1, the system sends the task description and available tools in a system prompt, followed by the user's input. In steps @2 and above, we enter the `ForEach` loop listing the history after the initial messages. The history is presented as alternating assistant and user messages, where the assistant messages list the requested tool and its reasoning at the given time in the history, and the user message contains the tool response.

Note that the very last step of the process, in which the LLM responds with the final answer which is returned to the user, is not part of the ACDL context: the system employing the `React1` context, will not send this information to the LLM, as it will produce the answer, finishing its job. ACDL describes what the LLM receives as context, not how the user experiences the system.

**Time steps and sub-steps.** ACDL's model of agentic systems is of discrete-time state machines, operating on their own clock. Each state change in the system is a clock-tick (step increase), but a clock may tick also without a state change. Clock ticks often correlate with actions, as actions change state (hence moving the system to the next step). However, it is possible that several actions will result in a single step change: for example when the system collects all the action results internally before it updates its state based on all of their outcomes. The outside world may of course progress regardless of the system's clock. But observing the outside world amounts to a clock tick. The system's clock may progress due to: (a) external events from the environment (like the arrival of user input; (b) internal timers for periodic checking of state ("hearbeats"); or (c) the conclusion of some internal process that necessitates a state change (the conclusion of some actions or tool calls).

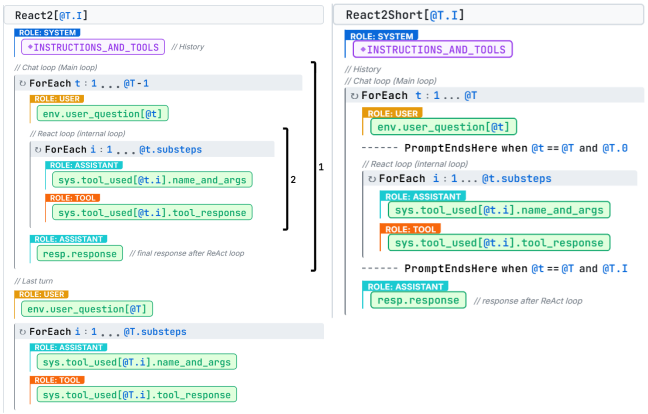
It is convenient to think of the system time-steps as being organized in a hierarchy. The top-level is what we consider to be the "main" steps. Major events that make the clock tick. This varies between system to system. For example, in a chat-bot system, user events trigger the system's operations, and hence the main time steps are driven by user inputs or their equivalent: each user input is a main step increase. In the React1 system above, there is only a single user input, which is then followed by a series of tool calls for obtaining an answer. Here, the main step is an iteration in the ReAct loop. But it is common for a system to work on both time scales. For example, `React2` below (left) describes a multi-turn, multi-step system, in which at each *turn* (outer loop, chat-loop) a user enters a query and awaits a response, but producing each response involves a series of *computation steps* (inner loop, ReAct loop).

This is handled by ACDL by introducing the concept of a sub-step, maintained by the system. The signature `React2[@T.I]` indicates the context at the *I*th substep of the *T*th main step.[4] For each step `@t`, `@t.substeps` holds the number of sub-steps that participated in it. We can then access sub-steps via `@t.i` as done in the inner loop (2) and the final loop (3).[5]

For nested loops, the context description may have multiple "exit points": here, at the main time steps `@T.0` the context concludes after the last user line, while for inner time-steps `@T.I > 0` the context concludes at the end. We can make this explicit with `PromptEndsHere` markers. Using them also helps to remove the trailing "last turn" in some cases (`React2Short`).

[4]Time steps can nest beyond two level, e.g., @T.I.J.K. Variable nesting levels are indicated by `@T.*`.

[5]By convention, the current time step is indicated with upper letters, and the indices iterating over its range in the lower-cased version of the same letter.




**Reuse through Fragments.** In some cases, the same context construction logic is used in multiple different LLM calls in the system, or even in different phases within the same LLM call. ACDL offers a mechanism called *fragments* to capture this commonality and avoid repetition. A fragment receives parameters as input and results in either a string or a list of role messages. The semantics of invoking a fragment from within a context definition or another fragment is that the invocation site is replaced with the fragment value (invoking fragments resulting in strings are valid only within role messages, and invoking ones expanding to a list of role messages is valid only outside of them). The fragment definition syntax is very similar to context definition syntax. See details and examples in Appendix D.

**Multi-agents.** Finally, ACDL can also represent contexts in multi-agent systems. In some agentic systems, e.g., as implemented in coding agents such as OpenCode, multiple agents are supported transparently: each agent is isolated from the others, having its own isolated state and clock, and "agent invocation" is handled as a tool calling. Every turn of the tool sub-agent is a tool-call for the invoker agent, and the sub-agent treats these calls as user inputs from the environment (where the user happens to be an agent).

In contrast, some systems require a shared clock or shared memory, or that agents approach each other by name. To this end, the ACDL `Context` definition can be parameterized not only by the time step, but also by a variable indicating its agent id. Then, elements such as `sys[agent].foo` refer to a property specific to `agent` agent, and `sys[agent].foo[@t]` refers to this agent property at time @t. `sys.foo` and `sys.foo[@t]` refer to shared state. Agents who have access to another agent's ID, can use it within their own context, to refer to the other agent's state. In such systems, it is convenient if all agents share the same clock, and @t refers to the same clock. ACDL may be extended to support shared state without shared clocks in the future.

Figure 3 shows an example ACDL-defined context that includes an agent parameter. The context includes a history of the agent's last 50 actions and their results, and a description of the agent's current inventory. For each of the agent's peers observed in its environment, the context includes the peer's name, description, and details retrieved from memory. If the agent is currently engaged in a conversation with another agent, the context also includes a history of that conversation as a sequence of turns, each consisting of the

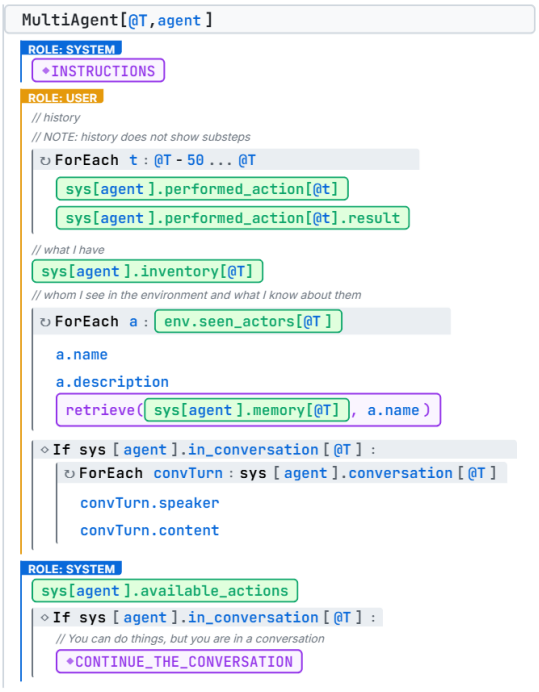



**Figure 3: ACDL Description of a simple multiagent main loop.**

speaker and what they said. Next, the agent is told what actions are available to it. Finally, if the agent is currently in a conversation, it is instructed to continue it.

## 5 ACDL at Work

The argument that a standard context description language would be helpful to the science and engineering of agentic systems is easy to make, as any standard language, if widely adopted, facilitates unambiguous communications, comparisons and analysis. In this section, we attempt to make the more challenging argument, that *ACDL is an excellent candidate standard for adoption.*

**ACDL has tools to ease adoption.** Tools facilitating the use of ACDL in describing contexts are made available (see www.acdlang.org). We provide a web-based editor and visualizer that renders ACDL specifications and allows to download them as PDF or PNG; This is accompanied by complete language documentation (formatted for both humans and coding agents), as well as a set of example `.acdl` specifications that users can explore to familiarize themselves with the language and use as basis for their own. Additionally, we offer a VS Code extension with syntax highlighting and live rendering of ACDL files directly within the editor, along with a Claude Code skill `.md` file for an integrated authoring experience. The resources are open source, and we welcome bug fixes, extensions, and contributions. Finally, ACDL also lends itself to whiteboard diagramming in face-to-face communication.

Beyond its utility for human-to-human communication, ACDL may also serve as a medium for unambiguous human-to-AI communication. We anecdotally observed that Claude Code could read ACDL specifications and produce corresponding agentic loops, as

well as convert from one agentic loop format to another. We flag this as a promising avenue for future work, and expect the underlying capability to improve rapidly as models progress.

**ACDL makes context nuances easy to locate and communicate.** Through its visual rendering and the use of labels and numbered annotations, ACDL facilitates clear communications between those explaining the evolving structure of a context, and those trying to understand or compare it to others. From the point of view of a reader looking to understand the differences between contexts, they are apparent at a glance. From the point of view of an author, trying to communicate important details or differences, the use of numbered annotations in the visual rendering allows references from within the accompanying text.

For example, we consider the simple multi-turn multi-step ReAct agent as implemented in the MINT benchmark [9] and documented in ACDL in Figure 4. We then attempted a trivial variant: we remove the lines marked as (1)—removing reasoning traces from the tool-call history. This variant hurt performance on reasoning tasks while slightly improving performance on the code tasks. We tried 5 other variants in the experiment; see A for more details about this experiment.

**ACDL helps document and clarifies complex contexts.** ACDL can go beyond simple examples, and scales to documenting elaborate contexts of game-playing systems, multi-agent systems and real-world state-of-the-art systems. We demonstrate it through examples: (1+2) documenting the main contexts of the popular OpenCode [6] and OpenClaw [7] systems, as reverse-engineered by us from a combination of digging through code and inspecting proxy traces; and (3) documenting the Gemini Pokemon Blue agent as described meticulously in a detailed technical report [4].

**OpenCode and OpenClaw.** OpenCode [6] and OpenClaw [7] are two very popular open-source projects implementing widely-used agentic systems. OpenClaw is a personal assistant agent which can access the user's computer and performs tasks on their behalf, while OpenCode is a strong-performing coding-agent similar to claude-code. How are the contexts of these two systems implemented? We traced (reversed engineered) the main loop prompts of the two projects by looking at the code as well as at LLM-call traces, and documented them with ACDL (Figures 5 and 6).

Both systems follow the familiar multi-turn multi-step ReAct-loop pattern, where the lifetime of the loop is a "session" and most of the logic is implemented via tool-calling. In both systems, sub-agents are exposed as tools to be called by the main agent, with completely isolated contexts and states, hence are seamlessly supported. OpenClaw has a very long system prompt which includes also skills, tools, sub-agents and memories (not documented here). For OpenCode, the system prompt is more minimal, delegating skills, tools and sub-agent definitions to the "tools" parameter. The system prompt does include basic environment constants: working directory, git repository status, platform, and session start date.

Both systems also make use of the LLM ability to request several tool calls in the same turn. In both systems the tool-call requests for a step are appear in the history within a single assistant message, while their responses are presented in a sequence of individual tool messages (marked as (1)).

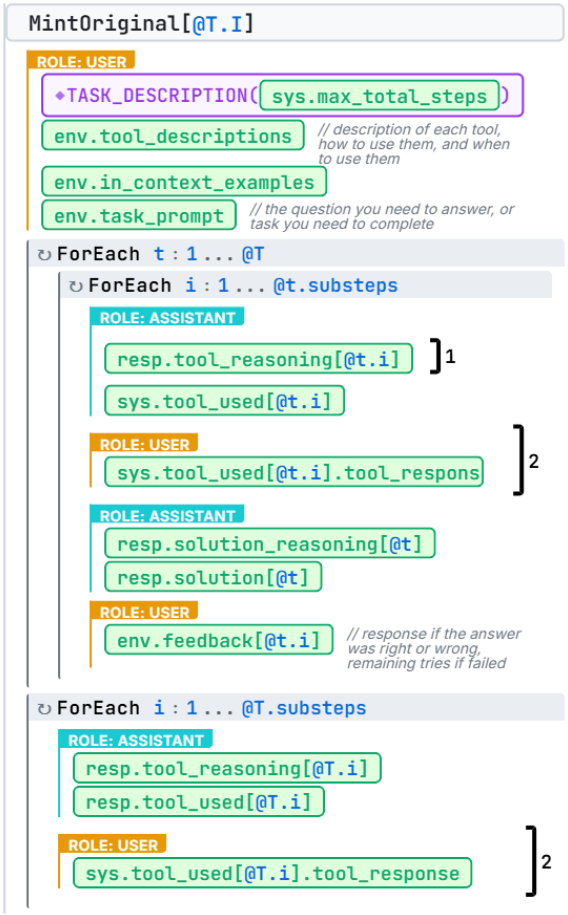



**Figure 4: ACDL descriptions of a multi-turn multi-step ReAct agent as implemented in MINT [9]. The markings indicate variants: removing the line marked in (1); replacing the role of the line marked (2) from *user* to *tool*.**

Both systems also support context compaction when the history grows too large. They differ somewhat in their summary presentation strategies (marked as 2): OpenClaw has an initial user message summarizing the previous conversation, followed by an assistant response to this summary. In contrast, OpenCode takes the opposite approach: an initial user message asks "what did we do so far" followed by an assistant message with the summary.

We also observe departures from and additions to the "classic" ReAct loop: OpenCode distinguishes between Plan-mode and Build-mode, and this is supported (marked as 3) by injecting "reminders" at the end of the last user message in the conversation. OpenClaw includes a time-stamp in each user input (marked 4), supports non-user initiated tasks using a Hearbeat signal implemented as special user-messages (marked 5) and sometimes injects pending async messages on top of user-message content (6). ACDL makes these design similarities, as well as differences, apparent, inspectable, and amenable for discussion and potential cross-pollination and mutual improvements.

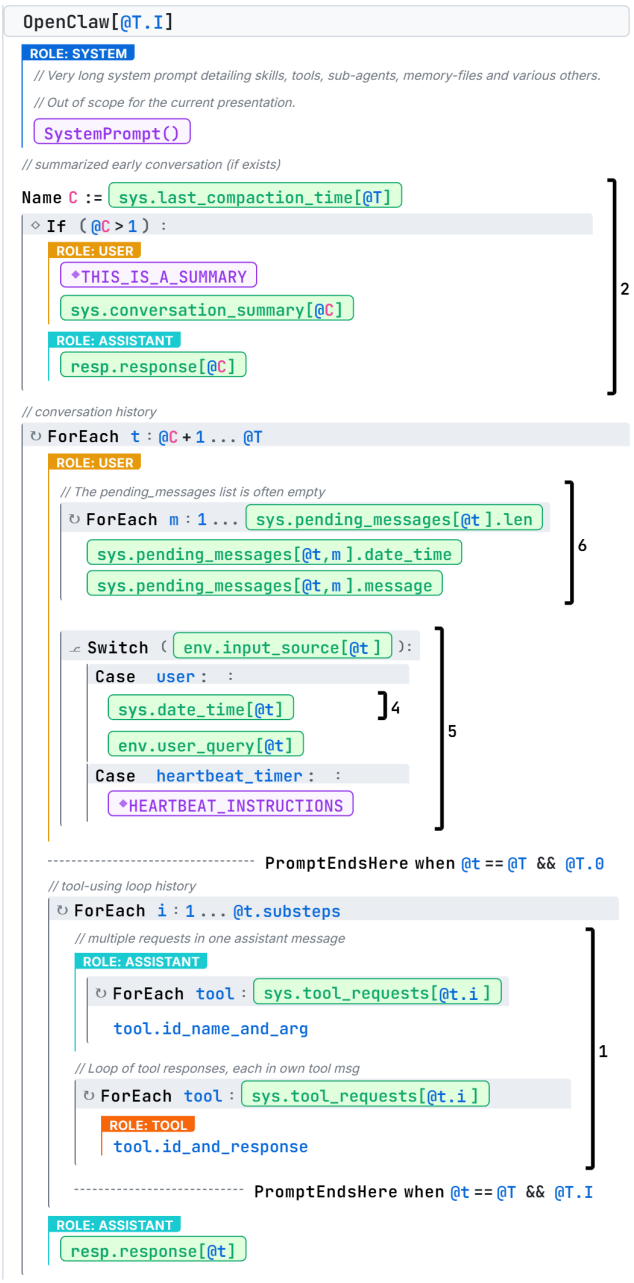



**Figure 5: ACDL Description of OpenClaw main loop, without System Prompt details.**

**Gemini Plays Pokémon Blue.** The Gemini 2.5 technical report [4] highlights the Gemini Plays Pokémon project [12] as a demonstration of the model's agentic capabilities, in which Gemini 2.5 Pro was set up as an autonomous agent to play Pokémon Blue from start to finish via a Twitch livestream, with the goal of beating the entire game (all 8 gym badges + Elite Four).

We include the verbatim description from the Gemini report in Appendix B; here we provide a brief overview in our own words. The agent receives observation information along with a system

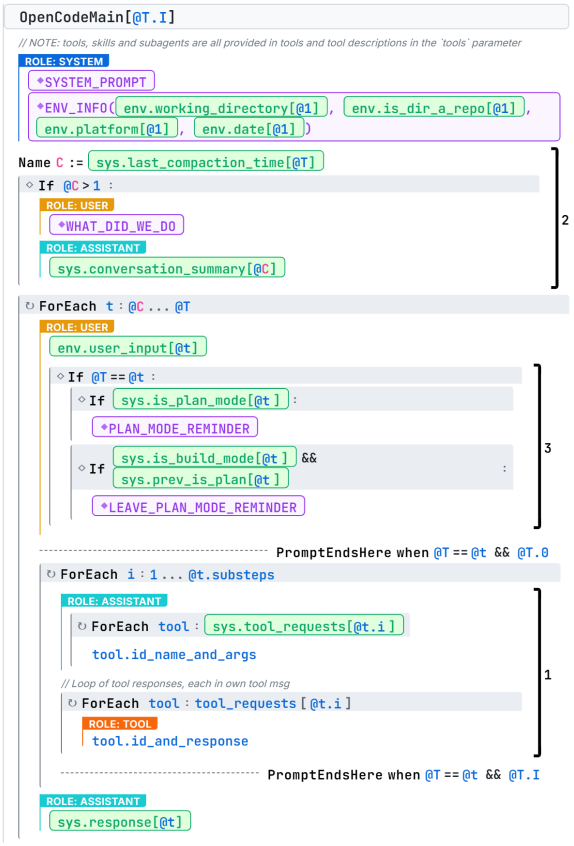



**Figure 6: ACDL Description of OpenCode main loop (tools, skills and sub-agents are passed in the *tools* API parameter).**

prompt explaining that it is playing Pokémon Blue and that its goal is to beat the game. It is also given two tools (named *Path Finder* and *Boulder Puzzle Strategist*) and some general game-play tips. The agent then receives a summarized history of its previous actions: the most recent actions are included verbatim, while older history is compressed into summaries. Every 25 turns, a sub-agent evaluates progress on three main goals and several sub-goals, and the resulting progress report is injected into the context. Finally, the agent is instructed to explore and choose an action.

The ACDL description of the Gemini report is much easier to understand and is less ambiguous, though it captures the same details as the report. The results in the visual rendition in Figure 7. It makes very clear what the history sent to the agent at each turn is, along with the data that it gets about the game and its goals. Additionally, the context specification makes use of a *function* to show a call to a subagent that critiques the agent's decision by reviewing the history of his moves.

## 6 Related Work

Several languages have been proposed to bring structure to prompt engineering, though their goals differ substantially from ACDL's.

PromptML [11] is a domain-specific language for writing individual prompts as structured code. It provides explicit sections for context, objectives, instructions, examples, and metadata, giving


**Figure 7: Gemini Plays Pokémon Blue context structure visualization in ACDL.**

prompt engineers a deterministic format for specifying what goes into a single prompt. PromptML focuses on the internal organization of one prompt rather than on how prompts evolve across interaction steps.

PDL (Prompt Declaration Language) [2] is a YAML-based declarative language developed by IBM for composing LLM calls with tool use, code execution, and data processing into executable pipelines. PDL is a programming language for building LLM-powered applications: its specifications are programs that run, call models, and produce outputs. Its concern is the orchestration of computation, not the description of context structure.

POML (Prompt Orchestration Markup Language) [13], developed by Microsoft, uses an HTML-like syntax with semantic tags such as `<role>`, `<task>`, and `<example>` to organize prompt components. It also provides data integration components for embedding documents, tables, and images, along with a CSS-like styling system for managing presentation. Like PromptML, POML is concerned with structuring the content of individual prompts rather than with describing how context changes over time.

ACDL differs from all three in both scope and intent. Where PromptML and POML focus on organizing the content within a single prompt, and PDL focuses on orchestrating executable LLM pipelines, ACDL is a language for describing how the full context is assembled and how its structure evolves across interaction steps. It is not designed to be executed or to produce prompts directly, but rather to make context construction strategies explicit, communicable, and comparable. Its core constructs—time indexing, control flow over interaction history, and cross-step references—address a concern that is largely absent from these other languages: the temporal dynamics of context in multi-turn agentic systems.

## 7 Discussion

The design of context structure and its evolution across multiple invocations is central to the performance of LLM-based agentic systems, yet the field lacks a standardized method for communicating the intricate details needed for reproducing contexts, pointing out details, and communicating differences between contexts used in different systems. To address this need, this paper introduced a candidate standard, the Agentic Context Description Language (ACDL). ACDL is a formal, human- and machine- readable, implementation-agnostic language that allows precise descriptions of complex context structures and their evolution across LLM invocations. ACDL descriptions are automatically rendered into visual descriptions.

***Limitations.*** We currently see two main limitations for ACDL. First, ACDL currently handles only agentic systems whose execution model is a sequence of (potentially nested) discrete time steps in which agent state mutates between LLM context constructions *but remains immutable within each construction.* While technically all agentic systems *can* be implemented this way, some real-world systems are not: they have mutable state that changes during context construction. Such systems are currently cumbersome to translate into ACDL descriptions, as they first have to be converted to an equivalent immutable-during-construction implementation and only then described. This friction is a real limitation, which we hope to mitigate in future versions.

Second, ACDL currently lacks a clean way to describe multi-agent systems in which the different agents have separate clocks (each agent runs according to its own time steps, without a synchronized clock), yet all have access to some *shared* mutable state that can be read and modified by any of them. Such systems are plausible even if uncommon today. The only current workaround in ACDL is an explicit clock synchronization mechanism invoked when accessing the shared mutable state, which is inelegant.

We plan to address both of these in a future version, and will appreciate suggestions on how to do so.

## Acknowledgments

We gratefully acknowledge financial support from ISF Grant #1373/24. As always, thanks to K. Ushi.

## References


[1] Vincent Abbott. 2024. *Neural Circuit Diagrams: Robust Diagrams for the Communication, Implementation, and Analysis of Deep Learning Architectures.* Technical Report arXiv preprint arXiv:2402.05424. arXiv.

[2] Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, et al. 2024. *PDL: A Declarative Prompt Programming Language.* Technical Report arXiv preprint arXiv:2410.19135. arXiv. https://arxiv.org/abs/2410.19135

[3] DeepSeek-AI. 2026. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence. https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro/blob/main/DeepSeek_V4.pdf. Technical report.

[4] Gemini Team, Google. 2025. *Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities.* Technical Report. Google. https://storage.googleapis.com/deepmind-media/gemini/gemini_v2_5_report.pdf

[5] Guy Marshall, André Freitas, and Caroline Jay. 2025. An Evidence-Based Guidance Framework for Neural Network System Diagrams. *PLOS ONE* 20, 3 (2025), e0318800. doi:10.1371/journal.pone.0318800

[6] OpenCode 2025. OpenCode: The Open Source AI Coding Agent. https://www.opencode.ai/. Open-source provider-agnostic coding agent for terminal, desktop, and IDE.

[7] Peter Steinberger. 2025. OpenClaw: Your Own Personal AI Assistant. https://www.openclaw.ai/. Open-source autonomous AI agent framework. Originally released as Clawdbot, November 2025.

[8] Knut Sveidqvist and Contributors to Mermaid. 2014. Mermaid: Generate diagrams from markdown-like text. https://mermaid.ai/

[9] Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2024. MINT: Evaluating LLMs in Multi-turn Interaction with Tools and Language Feedback. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR).* https://arxiv.org/abs/2309.10691

[10] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR).*

[11] Naren Yellavula. 2024. PromptML: Prompt Markup Language. https://www.promptml.org GitHub repository.

[12] Joel Zhang. 2025. The Making of Gemini Plays Pokémon. https://blog.jcz.dev/the-making-of-gemini-plays-pokemon Blog post.

[13] Yuge Zhang, Nan Chen, Jiahang Xu, and Yuqing Yang. 2025. *Prompt Orchestration Markup Language.* Technical Report arXiv preprint arXiv:2508.13948. arXiv. https://arxiv.org/abs/2508.13948

# Appendices

## A The MINT Experiment

We evaluated 7 context configurations that share the same system message structure—task description, tool descriptions, in-context examples, and task prompt—but differ along two structural axes: how much tool-use history is retained across turns, and how tool calls and their responses are grouped into messages.

We evaluate all configurations on the MINT benchmark tasks (excluding ALFWorld). The reasoning tasks are drawn from GSM8K, HotpotQA, MATH, MMLU, and TheoremQA, while the code-generation tasks are drawn from MBPP and HumanEval. We use GPT-4-Turbo as the underlying LLM. The results are summarized in the tables below.

**(a) Reasoning tasks**

| **Variation** | **Success Rate** (%) |
|---|---|
| var2 | 77.53 |
| mint-original | 76.90 |
| var6 | 76.90 |
| var1 | 74.68 |
| var4 | 74.37 |
| var5 | 74.05 |
| var3 | 72.78 |

**(b) Code generation tasks**

| **Variation** | **Success Rate** (%) |
|---|---|
| var6 | 54.41 |
| var3 | 54.41 |
| var4 | 54.41 |
| mint-original | 52.21 |
| var2 | 51.47 |
| var5 | 50.74 |
| var1 | 48.53 |

These choices clearly affect performance: the gap between the best and worst configurations reaches nearly 5 percentage points even in this simple setup, the best variation for reasoning differs from that for code generation, and the MINT-original formulation is not the top performer in either category.

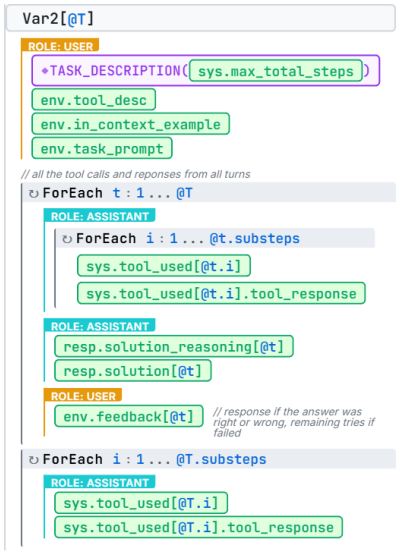
We used GPT-4-Turbo, a relatively weak model by current standards, because stronger models such as GPT-5 or claude-opus 4.6 are largely insensitive to these distinctions on MINT, where prompts are short, tasks are simple, and time horizons are limited. This insensitivity does not imply that context engineering no longer matters; rather, it reflects the fact that academic benchmarks with short interaction horizons fail to surface effects that would likely emerge in real-world, long-horizon agentic systems.

## B Gemini plays Pokémon blue

The following is taken verbatim from the Gemini 2.5 technical report [4], describing the Gemini Plays Pokémon agent [12]:

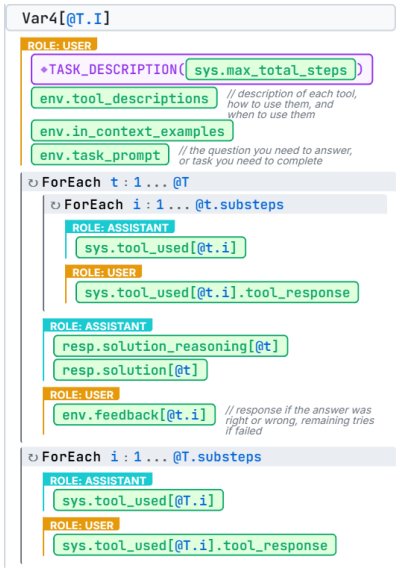
> The Gemini Plays Pokémon agent [12] receives a subset of RAM information, intended to give sufficient information to play the game, partially overlaid with a screenshot of the Game Boy screen. Gemini is prompted with a system prompt telling it that it is playing Pokémon Blue and that its goal is to beat the game, as well as descriptive information to help it understand the conventions in the translation from vision to text and a small number of general tips for gameplay. Gemini then takes actions, translated to button presses. The sequence of actions is stored in context, followed by a summary clear every 100 turns. The summaries are stored in context as well. Every 1000 turns GPP compresses the existing summaries again. Additionally, Gemini keeps track of three main goals (primary, secondary, and tertiary) as well as several additional goals (contingency plans, preparation, exploration, team composition). Every 25 turns, another prompted instance of Gemini (Guidance Gemini, or GG) observes the same context as the main Gemini and critiques performance and attempts to point out hallucinations and so on. The overworld fog-of-war map is stored in the context in XML, where coordinates which have not been seen cannot be viewed until explored. Crucially, in the system prompt, Gemini is instructed to explore. Once a tile is explored, however, the coordinate is automatically stored in the map memory and labeled with a visited counter. Tiles are also labeled by type (water, ground, cuttable, grass, spinner, etc.), and warp points to different maps are also labeled as such. Gemini also has access to two agentic tools, which are both instances of Gemini equipped with a more specialized prompt—the `pathfinder` tool, and the `boulder_puzzle_strategist` tool. In the `pathfinder` prompt, Gemini is prompted to mentally simulate a path-finding algorithm, which is left unspecified, and to verify that the path is valid against the map information available. In the `boulder_puzzle_strategist` tool, Gemini is prompted to solve special boulder puzzles that are present in Pokémon Blue in the Victory Road dungeon—these puzzles are similar to the game Sokoban—again, by mentally simulating sequences of actions that lead to solutions to the puzzle. The prompt describes the physics and the task of the boulder puzzle, as well as the desired output of solutions.


**(a) Variation 1. Previous turns include only the final reasoning and solution, omitting thinking and tool traces. The current turn includes the tool call in an assistant message and the tool response in a separate tool-role message.**



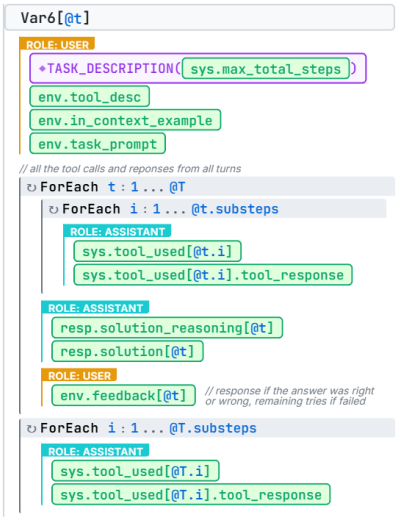

**(b) Variation 2. Previous turns include all tool calls and responses, with each turn collapsed into a single message. The current turn places each tool call and its response in their own messages.**


**(c) Variation 3. Previous turns include all tool calls and responses, with each tool call in a separate assistant message and each response in a separate tool-role message. The current turn is structured as in Variation 1.**




**(d) Variation 4. Same as Variation 3, but tool responses are placed in user-role messages instead of tool-role messages.**


**(e) Variation 5. Same as Variation 1, but the final tool response is merged into the preceding assistant message instead of appearing in its own tool-role message.**




**(f) Variation 6. Previous turns split each tool interaction across two messages: an assistant message for the tool call and a tool-role message for the response. The current turn is structured as in Variation 2.**

**Figure 8: Six variations of the Mint context explored in our experiments. Across all variations, the user instruction message and the reporting of previous turns' solutions, reasoning, and feedback remain unchanged.**

## C DeepSeek-V4

Figure 9 shows figure 7 from the tech report for the DeepSeek-v4 [3] models. This visual rendition has striking resemblance to the way ACDL is rendered, but ACDL also formally captures the dynamic evolution using the time (turn) steps @T, the separation of messages, and the role of each message.


**Figure 9: Figure 7 from the tech report of the DeepSeek-v4 [3] models.**

The following is taken verbatim from the DeepSeek-V4 technical report [3], which describes some of the context management strategies for their agent.

> DeepSeek-V3.2 introduced a context management strategy that retains reasoning traces across tool-result rounds but discards them upon the arrival of new user messages. While effective, this still caused unnecessary token waste in complex agentic workflows — each new user turn would flush all accumulated reasoning content, forcing the model to reconstruct its problem-solving state from scratch. Leveraging the expanded 1M-token context window of DeepSeek-V4 series, we further refine this mechanism to maximize the effectiveness of interleaved thinking in agentic environments:
>
> - **Tool-Calling Scenarios.** As illustrated in Figure 7(a), all reasoning content is fully preserved throughout the entire conversation. Unlike DeepSeek-V3.2, which discarded thinking traces upon each new user turn, DeepSeek-V4 series retain the complete reasoning history across all rounds, including across user message boundaries. This allows the model to maintain a coherent, cumulative chain of thought over long-horizon agent tasks.
> - **General Conversational Scenarios.** As illustrated in Figure 7(b), the original strategy is preserved: reasoning content from previous turns is discarded when a new user message arrives, keeping the context concise for settings where persistent reasoning traces provide limited benefit.

Figures 10 and 11 show the ACDL specifications for the context descriptions shown in Figure 9. Our figures also state what the role is assigned to each of the messages. This might not be accurate for this specific case because this information is missing in the tech report description and is inferred by us.age

```
DeepSeekAgentWithTools[@T.I]
ROLE: SYSTEM
  sys.tools
↻ ForEach t : 1 ... @T
  ROLE: USER
    env.user_message[@t]
  ↻ ForEach i : 1 , @t.substeps
    ROLE: ASSISTANT
      resp.thinking[@t.i]
    ROLE: ASSISTANT
      resp.tool_call[@t.i]
    ROLE: USER
      env.tool_response[@t.i]
  ◇ If I == 0 and @t > 1 :
    ROLE: ASSISTANT
      resp.answer[@t]
```

**Figure 10: ACDL description of the DeepSeek agent with tools**

```
DeepSeekAgentWithoutTools[@T]
ROLE: USER
  env.user_message[@1]
↻ ForEach t : 2 ... @T
  ROLE: ASSISTANT
    resp.answer[@t-1]
  ROLE: USER
    env.user_message[@t]
```

**Figure 11: ACDL description of the DeepSeek agent without tools**

# D ACDL: Formal Language Reference

This is a complete reference for the Agentic Context Description Language (ACDL) introduced in Section 4. The language declaratively specifies prompt structures sent to large language models, separating structural concerns from content and implementation.

**Specification Structure**

An ACDL specification defines a named prompt template parameterized by optional indices. The body consists of an ordered sequence of *prompt blocks*—role messages, label blocks, and control flow constructs—which may be freely interleaved:

```
PromptName[idx1, idx2, ...]: {
  <prompt-blocks>
}
```

A single file may contain multiple specifications, each defining a separate prompt template. Files may also contain *fragment definitions*—reusable building blocks that encapsulate portions of a prompt. Fragments come in two varieties: string fragments (`StrFrag`), which produce content pieces, and role fragments (`RoleFrag`), which produce complete role messages. These are described in detail in the Fragments section below.

```
StrFrag FragmentName[params]: {
  <content-elements>
}

RoleFrag FragmentName[params]: {
  <prompt-blocks>
}

PromptName[idx1, idx2, ...]: {
  <prompt-blocks>
}
```

**Role Messages**

Every message carries exactly one role. Four roles serve the chat format; a fifth pseudo-role serves the legacy completion format:

| Marker | Purpose |
|---|---|
| `S:` | System—instructions, persona, tool descriptions, behavioral constraints. |
| `U:` | User—external input, observations, tool results. |
| `A:` | Assistant—prior model outputs, reasoning traces, chosen actions. |
| `T:` | Tool—structured tool call results. |
| `N:` | None—single unstructured text block (completion format only). |

Role messages support two syntactic forms. The *multi-line* form encloses content in braces and permits any combination of content elements and control flow:

```
S: {
  INSTRUCTIONS
  AVAILABLE_TOOLS
  env.datetime[@t]
}
```

The *single-line* form omits braces and accepts exactly one content element—a context variable, template, or function call:

```
U: env.user_question[@t]
A: resp.answer[@t]
S: INSTRUCTIONS
```

Control flow constructs (`ForEach`, `If`, `Switch`) are *not* permitted in single-line role messages. To include control flow inside a role message, the multi-line braced form must be used:

```
U: {
  ForEach(item: env.items) {
    env.item_detail[@t, item]
  }
}
```

The `N:` role (completion format) imposes two additional constraints: (1) exactly one `N:` block may appear per prompt, and (2) no chat roles (`S:`, `U:`, `A:`, `T:`) may appear in the same specification:

```
CompletionPrompt[@t]: {
  N: {
    TASK_DESCRIPTION
    env.context[@t]
    QUESTION
  }
}
```

**Scoping Rules**

The language enforces a strict two-level scope that mirrors the structure of LLM chat APIs. At the *top level* (the prompt body, outside any role block), only role messages, label and marker blocks, control flow constructs, name definitions, fragment invocations (role fragments), comments, and fragment definitions are permitted. *Inside* a role block's braces, valid elements are context variables, functions, templates, control flow, comments, name definitions, fragment invocations (string fragments), marking blocks, and `break`/`continue`. Role messages may **not** appear inside other role messages—the following is invalid:

```
U: {
  S: INSTRUCTIONS   // ERROR: nested role
}
```

For completion prompts using `N:`, the top level may contain only the single `N:` block—no other role messages, Mark blocks, or control flow may appear outside it.

**Context Variables**

Context variables reference dynamic runtime data. The general syntax is:

`namespace.path[indices]`

where `namespace` is one of three reserved prefixes:

| NameSpace | Use Cases |
|---|---|
| `env` | Environment—external inputs, observations, sensor readings, user queries, game state. |
| `sys` | System—agent state, memory contents, tool configurations, action histories. |
| `resp` | Response—prior LLM outputs, reasoning traces. |

Paths may be nested using dot notation to reach into sub-fields of structured data. Indices (described below) may appear at any level of the path:

```
env.user_question[@T]  // the user question at time t
sys.agent_desc // the agent description (constant, does not change over time)
sys.tool[@t].tool_response[@t]  // the tool response of tool at time t.
env.bomb_location[@T, bomb] // the bomb location of bomb at time T
```

A variable without indices (e.g. `sys.agent_desc`) refers to data that does not vary over time or other dimensions.

**Indices**

Indices address specific elements along one or more dimensions. Two types are distinguished syntactically.

The special symbol `@` denotes the primary time dimension that the prompt iterates over. What `@` represents depends on the agent's structure: for a ReAct agent that operates within a single turn but loops over many steps, `@` refers to the current step; for an agent that loops over multi-turn conversations, `@` refers to the current turn, and sub-steps within each turn are indexed with ordinary index variables.

The current time is denoted with capital letters: `@T` for the main time step, and `I`, `J`, etc. for sub-steps. When iterating over time steps, the corresponding lower-case letters are used. Sub-steps are accessed using dot notation: `@t.i` refers to sub-step `i` within turn `t`, while `@T.I` refers to the current sub-step of the current turn. When iterating over the sub-steps of a previous turn, use `@t.substeps` to obtain the count. For example:

```
@T            // Current time step
@T-1          // Previous time step
@T.I          // Current substep of current turn
@t.i          // Substep i of turn t (in loops)

// Iterating over all previous turns
ForEach(t: range(1, @T)) {
  env.observation[@t]
}

// Iterating over substeps in the current turn
ForEach(i: range(1, I)) {
  sys.action[@T.i]
}

// Nested: substeps within each previous turn
ForEach(@t: range(1, @T)) {
  ForEach(i: range(1, @t.substeps)) {
    sys.action[@t.i]
  }
}
```

*Non-time indices* have no prefix and address other dimensions—named entities, or context-variable-valued keys:

```
[sys.agent_name]
[bomb]
```

Multiple indices are comma-separated: `env.bomb_location[@t, bomb]` addresses a specific bomb at a specific time step. Standard arithmetic operators (`+`, `-`, `*`, `/`, `%`) are permitted in all index positions, enabling expressions such as `@t-1`, `@t+1`, `t-k`, or `@t % 25`.

**Templates**

Templates are `ALL_CAPS` placeholders representing text blocks whose content is specified at instantiation time. They describe the semantic purpose of a text section without fixing its wording, separating prompt architecture from prompt prose. Words within a template name are separated by underscores: `TASK_INTRO`, `MAP_DESCRIPTION`.

Templates may accept arguments in parentheses, enabling parameterized text:

```
INSTRUCTIONS        // Task explanation
AVAILABLE_TOOLS     // Tool list
QUERY(sys.agent_name)  // Depends on the agent's name
```

An optional inline comment (after `//`) documents the intended content of the template.

**Functions**

Functions represent computed content—summarization, retrieval, formatting, or any transformation that cannot be expressed as a simple variable lookup. They are declared by name and purpose without defining their implementation; the name conveys semantic intent.

The syntax is `functionName(arg1, arg2, ...)[indices]`. The function names use `camelCase` (distinguishing them from `ALL_CAPS` templates). Arguments may be context variables, time or regular indices, numeric literals, arithmetic expressions, or nested function calls:

```
summarize(sys.history[@t])
get_dialog_history(sys.agent_name)
range(1, @T, 2)
```

The built-in `range(start, stop, step)` function generates numeric sequences for use in `ForEach` loops. The `step` argument is optional and defaults to 1. The range is exclusive, meaning `range(1, 3)` starts at 1 and ends at 2-1, excluding 3.

**Control Flow**

Three constructs govern dynamic prompt structure. All three may appear both at the top level (producing or gating entire role messages) and inside role blocks (controlling content within a single message).

*ForEach* iterates over ranges or collections to produce repeated structures. The syntax is:

```
ForEach(variable: iterable) {
  <body>
}
```

The iterable may be a `range(start, stop, step)` call or a collection-valued context variable. At the top level, the loop body may contain role messages, producing multiple messages per iteration. Inside a role block, it produces repeated content elements:

```
// Top-level: produces role messages
ForEach(@t: range(1, @T)) {
  U: env.user_question[@t]
  A: resp.answer[@t]
}
```

```
// Inside a role: produces content
U: {
  ForEach(bomb: env.bombs) {
    env.bomb_location[@t, bomb]
    env.bomb_details[@t, bomb]
  }
}

// Collection iteration
ForEach(agent: sys.agent_names) {
  U: env.utterance[@t, agent]
}
```

*If / ElseIf / Else* conditionally includes or excludes blocks based on runtime state. Conditions may use comparison operators (==, !=, <, >) and logical connectives (&, |):

```
If @T > 1 {
  ForEach(t: range(1, @T)) {
    U: env.user_input[@t]
    A: resp.answer[@t]
  }
}

If sys.tool[@t] == get_clarification {
  U: env.user_input[@i]
}
ElseIf sys.tool[@t] == search {
  A: env.search_results[@t]
}
Else {
  A: sys.tool_used[@t].tool_response
}
```

*Switch / Case / Default* selects among multiple alternatives based on the value of an expression:

```
Switch sys.action_type[@t] {
  Case "search" {
    U: env.search_results[@t]
  }
  Case "calculate" {
    U: env.calculation[@t]
  }
  Default {
    U: env.fallback[@t]
  }
}
```

The `break` and `continue` keywords are available inside loops with their standard semantics.

**Early Termination**

The `PromptEndsHere when` construct signals that, if the given condition is true, the prompt ends at that point—no further content is appended to the message sequence sent to the LLM for that turn. This is useful when certain conditions require a truncated prompt, such as an initial turn that needs no history or context beyond the setup. The syntax is:

```
PromptEndsHere when <condition>
```

For example, to end the prompt at the first sub-step of the current turn:

```
Prompt[@T]: {
  S: INSTRUCTIONS
  U: env.user_input[@T]
  PromptEndsHere when (@T == @t && @T.0)
  ForEach(i: range(1, @t.substeps)) {
    A: resp.answer[@T.i]
    U: env.feedback[@T.i]
  }
}
```

Here, if the current time is the first substep of the current turn, the prompt contains only the system instructions and user input. Otherwise, the full history of substep interactions is appended.

**Fragments**

Fragments are reusable building blocks that encapsulate portions of a prompt specification. They enable modular prompt design by allowing common patterns to be defined once and invoked multiple times. Two kinds of fragments are supported, distinguished by what they produce when expanded.

*String Fragments* produce content pieces without an associated role. They are defined with the `StrFrag` keyword and may contain any elements valid inside a role block—context variables, functions, templates, control flow, other string fragments, name definitions, and comments. When invoked, the content expands in place and inherits the role of the enclosing message:

```
StrFrag DocumentContext[doc]: {
  env.doc_title[doc]
  env.doc_content[doc]
  summarize(env.doc_metadata[doc])
}

StrFrag ConversationContext[@T]: {
  CONTEXT_HEADER
  ForEach(@t: range(@T-5, @T)) {
    sys.Summary[@t]
    If sys.has_tool_call[@t] {
      sys.tool_response[@t]
    }
  }
}
```

String fragments are invoked with the `Frag` keyword followed by the fragment name and arguments. They may appear anywhere a context variable, function, or template is valid—inside role blocks, within control flow bodies, or as arguments to other constructs:

```
U: {
  TASK_INSTRUCTIONS
  ForEach(doc: env.documents) {
    Frag DocumentContext[doc]
  }
  env.user_question[@T]
}
```

*Roles Fragments* produce one or more complete role messages. They are defined with the `RolesFrag` keyword and may contain role messages, control flow, mark blocks, and other prompt-level constructs—the same elements valid at the top level of a prompt body:

```
RolesFrag ConversationTurn[@t]: {
  U: env.user_input[@t]
  A: resp.answer[@t]
  If sys.tool[@t] != none {
    T: sys.tool[@t].tool_response
  }
}
```

Roles fragments are invoked at the top level of a prompt, wherever a role message would be valid. They expand to the full sequence of role messages defined in the fragment body:

```
ChatAgent[@T]: {
  S: INSTRUCTIONS
  ForEach(@t: range(1, @T)) {
    Frag ConversationTurn[@t]
  }
  U: env.user_input[@T]
}
```

Both fragment types accept parameters, enabling parameterized reuse. Parameters follow the fragment name in square brackets and may include indices, context variables, or other valid index expressions. The same invocation syntax (`Frag Name[args]`) is used for both types; the parser determines which kind based on context—invocations inside role blocks resolve to string fragments, while those at the top level resolve to role fragments.

Fragment definitions appear at the top level of an ACDL file, alongside prompt specifications. A single file may contain any combination of prompts and fragment definitions:

```
StrFrag ToolDescription[tool]: {
  sys.tool_name[tool]
  sys.tool_schema[tool]
}

RolesFrag ToolResult[@t, tool]: {
  A: sys.tool_call[@t, tool]
  T: sys.tool_response[@t, tool]
}

ToolAgent[@T]: {
  S: {
    INSTRUCTIONS
    ForEach(tool: sys.available_tools) {
      Frag ToolDescription[tool]
    }
  }
  ForEach(@t: range(1, @T)) {
    U: env.observation[@t]
    Frag ToolResult[@t, sys.selected_tool[@t]]
  }
  U: env.observation[@T]
}
```

**Mark Blocks**

A mark block annotates a section of the specification for visual emphasis in the rendered output. It places a bracket (`]`) along the side of the marked content, with a number identifier displayed beside it. Marks are purely presentational—they do not affect prompt semantics or scoping. They can wrap any prompt block, from a single content element to a large multi-message section:

```
Mark 1 {
  <prompt-blocks>
}
```

The number appears next to the bracket in the visualization as `]1`. Multiple marks with different numbers may be used to highlight distinct sections:

```
Prompt[@T]: {
  Mark 1 {
    S: {
      INSTRUCTIONS
      AVAILABLE_TOOLS
    }
  }
  Mark 2 {
    ForEach(@t: range(1, @T)) {
      U: env.user_question[@t]
      A: resp.answer[@t]
    }
  }
  Mark 3 {
    U: env.user_question[@T]
  }
}
```

Here, mark `]1` highlights the system setup, `]2` spans the conversation history loop, and `]3` marks the current query.

**Name Definitions**

A name definition binds a symbolic name to an expression, allowing complex or frequently repeated expressions to be written once and referenced concisely throughout the specification. The definition syntax is:

```
Name var_name := expression
```

Once defined, the name is referenced using the `$` prefix: `$var_name`. Name definitions improve readability by replacing long or opaque expressions with descriptive identifiers. The bound expression can be any valid ACDL element—a context variable, function call, arithmetic expression, or string literal. Fields of the bound value can be accessed via dot notation on the reference:

```
BasicRAG[@T]: {
  S: INSTRUCTIONS
  U: {
    Name docs := k_relevant_docs(env.user_input[@T])
    ForEach(i: range(1, $docs.len)) {
      $docs[i].source
      $docs[i].content
    }
    ANSWER_QUESTION_BASED_ON_DOCS
    env.user_input[@T]
  }
```

```
ToolAgent[@T]: {
  S: {
    INSTRUCTIONS
    AVAILABLE_TOOLS
  }
  U: {
    env.user_input[@1]
    env.user_document[@1]
  }
  If t > 1 {
    ForEach(@t: range(1, @T)) {
      If sys.tool[@t] == get_clarification {
        U: env.user_input[@t]
      }
      Else {
        A: sys.tool[@t].tool_response
      }
    }
  }
  S: {REACT_INSTRUCTIONS}
}
```

**Figure 12: Tool-using agent. System messages frame the task; a conditional loop replays interaction history with role assignment determined by action type.**

```
}
```

Here, `$docs` binds the result of a retrieval function, avoiding repetition of the full function call. Its length and individual elements are then accessed via `$docs.len` and `$docs[i]`.

Name definitions also support list comprehensions, which construct a list by iterating over a range or collection:

```
Name relevant_summaries :=
  [sys.summary[@t] for t in range(@T, @T-900, 100)]
compress_summaries($relevant_summaries)
```

This binds `$relevant_summaries` to a list of summaries sampled every 100 steps, which is then passed as an argument to a function. The reference syntax is the same regardless of whether the bound value is a single expression or a list.

**Comments**

Comments use `//` syntax and may appear on any line—between prompt blocks, inside role blocks, or between specifications. Once a `//` is encountered, the remainder of that line is treated as a comment; no further ACDL elements may follow on the same line. An *inline comment*, placed after a content element, renders alongside that element. A *standalone comment*, placed on its own line, renders at the current level of nesting:

```
S: {
  // This comment appears on its own line,
  // indented to the level of the S: block
  INSTRUCTIONS  // Renders beside INSTRUCTIONS
  AVAILABLE_TOOLS
  // Another standalone comment
  env.datetime[@T]  // Renders beside datetime
}
// This comment is at the top level
ForEach(@t: range(1, @T)) {
  // This comment is inside the loop body
  U: env.user_question[@t]  // Beside the message
  A: resp.answer[@t]
}
```

**Identifiers and Naming Conventions.** Identifiers start with a letter or underscore and may contain letters, digits, and underscores. They are case-sensitive. The language enforces naming conventions to visually distinguish element types: templates use `ALL_CAPS`, functions use `camelCase`, and context variable paths use `dot.separated.names`.

**An Illustrative Example**

Figure 12 presents a complete specification for a tool-using agent with conditional history replay.