COCORELI: Cooperative, Compositional Reconstitution & Execution of Language Instructions

Anonymous ACL submission

Abstract

We present COCORELI, a hybrid agent framework designed to address the limitations of large language models (LLMs) in tasks that require following complex instructions, minimizing hallucination, and various forms of reasoning. COCORELI uses medium-sized LLM agents with novel abstraction tools and discourse module to parse instructions to in-context learn dynamic, high-level representations of the environment. Experiments on natural collaborative construction tasks show that COCORELI outperforms single-LLM CoT and agentic LLM systems, all using larger LLMs. It largely avoids hallucinations, identifies missing information, asks for clarifications, and learn new objects. Using ToolBench API data, we tested whether models can reuse abstracted API sequences; COCORELI succeeded, while CoT baselines did not.

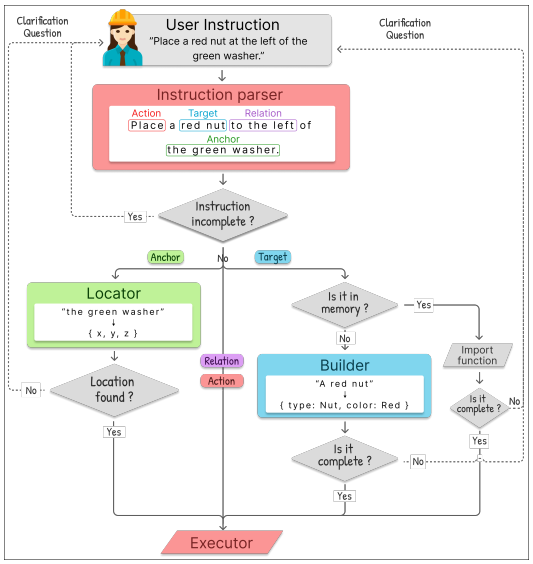


Figure 1: COCORELI, our hybrid agentic system.

1 Introduction

While Large Language Models (LLMs) have great appeal, real-world tasks often lack the large amounts of training data required to fine-tune LLMs on a specific task, and users must rely on few-shot learning or in-context learning. Moreover, zero- and few-shot LLM prompting consists of one input and output, which may hamper LLM performance especially when planning or multiple tools are required (Sypherd and Belle, 2024).

This has led to a strategy of decomposing complex tasks into subtasks and training agentic models to solve these subtasks. We develop and test a hybrid agentic approach for collaborative tasks. We illustrate its effectiveness on novel, complex assembly tasks in a 3D environment and also on Toolbench API completion. Our system COCORELI (Co-operative, Compositional Recon-struction & Execution of Language Instructions) is a modular and general architecture with the following novel features: (i) a top-level parser agent that interacts with the user to acquire instructions about a target structure, its properties, and task relevant properties; (ii) subsequent agents that identify what the target structure is and how to specify the task relevant properties; (iii) a memory component retrieves task relevant properties for familiar objects and stores newly acquired information; (iv) COCORELI learns new, complex object functions by abstracting over parameters in the located object derived from interactions with the user; (v) COCORELI uses a moderate sized LLM.

In its assembly task version, COCORELI builds objects with different primitive parts, colors, and at various locations. COCORELI's abstraction tool allows it to learn new constructs from one interaction. Each agent can ask a clarification question to the user for missing information, which it can then append to the instruction history. This reduces hallucinations. COCORELI's output is a JSON object for a situated construction.

The assembly task uses the ENVIRONMENT[1]

[1]We anonymize the name for the reviewing process.

framework, in which a human instructs the system to build simple or complex objects. We evaluate COCORELI and two other systems: (1) a Chain-of-Thought (CoT) approach that employs one large LLM with a single prompt to complete the task and (2) an agentic LLM with a large LLM (Llama-3.3-70b) handling subtasks with specific prompts. We create original data for each of our evaluation tasks and show that COCORELI outperforms options (1) and (2) on ENVIRONMENT tasks with less computational resources.

We also test COCORELI on ToolBench (Qin et al., 2024), a benchmark for tool learning models. Its abstraction and generalization ability improves over single-LLM CoT approaches.

COCORELI is a modular architecture that serves as a diagnostic framework to show the limits of current LLMs. Its general parsing, clarification, and memory modules, which are generic to most if not all interactive tasks, pinpoint failings of single-shot and agentic baselines, and why underspecified or compositional instructions remain challenging. Our controlled task-based ablations detailed in Appendix B show that each module is structurally necessary, while efficiency and burden analyses (see Appendix D) quantify the trade-offs of relying on general-purpose LLMs.

2 Related Work

COCORELI draws on many threads in current AI research, prompting techniques and in particular chain of thought(Wei et al., 2022; Wang et al., 2023), work on relating language and action, and tool learning. A discussion of this can be found in the Appendix C. We concentrate on the motivation for our modular design and comparison with modular and agentic frameworks.

Agentic systems incorporate multiple agents using LLM prompts. The workflow in such a system involves intermediary stages, resulting in better performance, adaptability, versatility, interpretability, and reliability in downstream tasks (Sudarshan et al., 2024). Agentic models may incorporate components that use and learn deterministic functions in context in a *hybrid* or *tool learning approach* (Schick et al., 2023; Qin et al., 2024).

Recent work has proposed several modular frameworks that combine reasoning, acting, memory, and clarification. ReAct (Yao et al., 2023b) interleaves reasoning traces with action steps, but assumes access to deterministic external APIs or tools; directly applying it in ENVIRONMENT is difficult since many instructions are under-specified and require structured clarifications rather than trial-and-error search. Reflexion (Shinn et al., 2023) improves exploration in reinforcement learning by iteratively revising failed trajectories, but lacks a mechanism for structured clarification in cases such as selecting correct part attributes (e.g., color of a nut). ART (Paranjape et al., 2023) decomposes tasks into tool-use programs by selecting reasoning templates, but presumes known tool schemas (e.g., search, calculator) and does not adapt well to environments with novel vocabulary and evolving action types. SayCan (Ahn et al., 2022) integrates LLMs with RL to ground instructions in robotic affordances, but requires extensive interaction data and fine-tuning, which ENVIRONMENT explicitly avoids by working in a low-data regime. SwiftSage (Lin et al., 2023) uses multi-agent debate with retrieval to improve decision-making in text-based environments with predefined templates, but does not involve physical grounding or structured spatial constraints. AgenticLU (Zhuang et al., 2025) tackles long-context reasoning by generating self-clarification questions, but operates purely in textual domains and cannot handle spatial underspecification or grounding. MapAgent (Kong et al., 2025) improves mobile task automation via trajectory-constructed memory and coarse-to-fine planning in GUI environments, but focuses on app navigation rather than multi-agent collaboration with physical constraints. Together, these frameworks highlight complementary directions in agentic LLM research, but none address the unique combination of underspecified instructions, structured spatial reasoning, and abstraction-based generalization that ENVIRONMENT requires. Importantly, such scenarios are not niche: real-world tasks such as assembling furniture from incomplete manuals, robotic assistants collaborating with humans in factories, or AI-powered CAD systems supporting engineers all involve vague or evolving specifications that must be clarified and grounded before execution.

Collaborative tasks Approaches to object identification with LLMs and reinforcement learning have proved that the task is complex and challenging to solve, and additionally ignore the needed human feedback (e.g. Kanitscheider et al., 2021; Lin et al., 2022). Voyager (Wang et al., 2024) is a learning agent based on GPT-4 that learns to play the

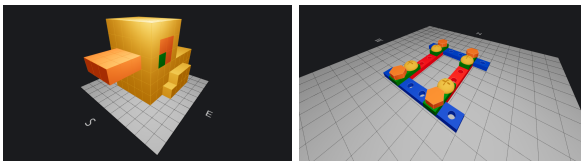
Minecraft game, relying on an iterative prompting mechanism to create functions for game milestones and on in-context learning to create new skills on the fly and store them in an external database for later use. However, Voyager provides limited interaction between a human and the model, which is a crucial component in a collaborative learning environment, and also overspecifies its stored skills. For instance, the skill "crafting a wooden axe" is a separate skill from "crafting a metal axe" and must be learned and stored separately. See Appendix C for further related work.

3 The ENVIRONMENT task

ENVIRONMENT is patterned on the Minecraft Collaborative Building Task (Narayan-Chen et al., 2019), which is a task between two human players, an Architect and a Builder. As in the Minecraft Collaborative Task (Narayan-Chen et al., 2019), the ENVIRONMENT task involves two agents in a collaborative task of building an object, each with different roles. The Architect has a plan for accomplishing the task and must teach the other agent (Builder) how to achieve it, since Builder lacks the plan.

While both environments share a common Architect-Builder setup, ENVIRONMENT introduces novel objects and scenarios that LLMs have not been trained on, and therefore must learn in-context. It also has features that make the task harder (see Figure 2 for a visual comparison). The larger grid size ($16 \times 16 \times 16$ in ENVIRONMENT vs. $11 \times 11 \times 11$ in Minecraft), increases the complexity of spatial reasoning. Second, unlike Minecraft, ENVIRONMENT features 9 basic objects: *nuts* (3 types), *washers*, *screws*, *bridges* (2 types), *bolts*, and *gaskets*. Different ENVIRONMENT parts can have different types of 3-D spatial coordinates. For example, bridge parts occupy two horizontal or vertical spaces (horizontal or vertical bridges), while other parts occupy only one space. This introduces additional constraints and complexity to decision-making, as the model must determine where to place a "two place" part given only some desired coordinates.

Finally, ENVIRONMENT introduces challenges that reflect real-world constraints in structured assembly, multi-agent collaboration, and planning under physical laws. Unlike Minecraft, which allows unconstrained placement of blocks, ENVIRONMENT enforces strict physics rules (e.g., gravity constraints), requiring agents to reason about stability and structural integrity. The gravity constraint directly impacts construction strategies, thus creating more opportunities for failure in ENVIRONMENT.



Minecraft ENVIRONMENT

Figure 2: Graphical representations of the Minecraft and the ENVIRONMENT environments.

4 COCORELI, our hybrid agentic system

Motivation Our first experiments on the ENVIRONMENT task revealed that using a single LLM with a long prompt led to frequent failures in retaining and applying critical information. We then explored a two-module setup for the construction scenario with a *Builder* and a *Locator*, which reduced individual prompt size and slightly improved performance, but still struggled in cases where instructions were vague or underspecified. Embedding conversational structures like clarification questions directly into the Builder further exacerbated prompt bloat, reintroducing earlier reliability issues. These limitations motivated COCORELI's modular three-agent architecture: an *Instruction Parser*, *Builder*, and *Locator*. The Instruction Parser isolates the selection of relevant shape templates from the instruction, enabling more focused prompts for the Builder and more reliable clarification handling. This modular separation significantly improved both parsing accuracy and robustness to missing information.

The final COCORELI architecture has several agents (Figure 1): **Discourse Module, Instruction Parser, Builder, External Memory, Locator**, and **Executor**. The agents are built with the LLaMA-3.1 8b LLM (Grattafiori et al., 2024). We describe them and their features below.

COCORELI's architecture contains a special feature, a **discourse module** (represented as a diamond in Figure 1), that generates clarification questions when a called function is missing information needed for execution. An effective approach to coding this type of problem is to represent each object as a JSON structure containing all its specific characteristics, initially set to *null*. As the user

provides information, these fields are progressively filled. This method is advantageous because LLMs often struggle to avoid hallucinating the necessary information. When attributes remain unspecified, the system requests more information via **clarification question** and iterates this procedure until the missing information is specified. Thus, the system combines agent actions iteratively to resolve ambiguities and fill in missing information. Our system collects and avoids hallucinating a lot.

Another feature of COCORELI is that agents can use an **abstraction function** over instantiated parameters in instructions or stored objects to acquire general plans. For example, given the input "*a tower made of three red nuts at the first column, second row*", COCORELI extracts the features and the location of each object separately and stores them. On a future step, the system is capable of replicating the multi-object structure in a different location or with different parts and colors.[2]

Instruction Parser The Instruction Parser interacts with the user to extract information from the user's instructions. In ENVIRONMENT, it searches for the objects' properties and locations from the textual input. The Parser passes relevant information to two other agents, the **Locator** and the **Builder**, which work in parallel.

Locator The Locator uses the information collected from the Instruction Parser to find the coordinates in the grid where the user wants to place one or more objects. If the coordinates are incomplete, the agent activates the clarification loop in the discourse module and asks for the missing coordinates.

Builder The Builder uses the given instructions to check with an **external memory** component if the part or structure mentioned in the instruction is a known entity or not. If the input refers to a known structure, it retrieves a JSON object representation from the memory and fills in the features specified in the instructions. If the input refers to a new object and the instructions are complete, the Builder uses those instructions to make the structure. Otherwise, it invokes the discourse module to ask for clarification.

External memory The memory stores hard-coded ENVIRONMENT functions and previously created shapes as relational graphs, with nodes as parts and edges encoding relative spatial positions. In addition to storing single-situated objects, COCORELI can use the abstraction function on a concrete situated object O in memory to construct an abstract function in which the instantiated parameters of O are abstracted away. This allows the system to recreate a version of O in any size, color, or other specifications. Another benefit is that our function-based approach is size-agnostic, meaning that we use one function for a complex structure instead of n placement instructions for the n components of the structure.

Executor The Executor combines information from Builder and Locator to yield a JSON object, with arguments corresponding to object properties (e.g., part type, color). The executor can then run this object as a deterministic program.

5 Experimental setup

5.1 Tasks and baselines

We set up the following ENVIRONMENT-related tasks for evaluating COCORELI and baselines:

(i) (Fully-specified) single-sentence instructions to place a single part
(ii) (Fully-specified) sequences of instructions
(iii) (Fully-specified) sequences to build complex shapes
(iv) Underspecified instructions (for one or two parts) and discourse ways to complete them
(v) Instructions to abstract from previous instruction sequences and to learn new functions from context to apply in new situations

Task (i) requires that the model parse the user input and extract all the relevant information for object properties and location. In this task, there is no need to retain previous information. Task (ii) additionally requires serially parsing multiple objects. Task (iii) demands the creation of a complex structure composed of simple and complex parts; COCORELI achieves this composition through a sequential call of functions. Task (iv) probes the model's ability to deal with missing information from user input; COCORELI uses the clarification loop to find the missing information. Task (v) is a continuation of Tasks (ii) and (iii), in which the user describes a complex shape that the system generates and then abstracts onto to get a general plan that can be executed in different ways. This task not only tests the system's memory, but also its capacity to abstract and learn from past specific

[2] We apply scaling, using Nearest Neighbor Scaling (Jenkins et al., 1985) to generate size variants of the stored structure.

instructions (e.g., place the same complex shape in different colors). These tasks also serve as *Task Based Ablation experiments*. More details are in Appendix B

Baselines As our first baseline, we test a **single LLM** with a CoT prompt. The model is prompted once and is asked to generate a reasoning chain and a final answer on the ENVIRONMENT tasks. This setup lacks an external memory; so for tasks requiring memory, we must repeat the instructions(s) that described the shape in the current prompt. In this single LLM setup, we test Claude 3.5 Sonnet (Anthropic, 2024) and GPT-4.1 (OpenAI, 2025). We also tested LLaMA-3.1 8b (Grattafiori et al., 2024) in preliminary experiments, but the model's accuracy was too low for a meaningful baseline.

As a second baseline, we use an **agentic LLM approach**. We prompt the LLM to generate a Python function that abstracts over an input sequence of placement instructions. The goal is to induce a reusable, parameterized function that captures the underlying pattern in the data. If the converted code is correct, recreating the code will reduce to just calling the created Python function. We use LLaMA 3-70b (Grattafiori et al., 2024) for the agents of this approach. For more software and hardware details of the baselines and COCORELI, see Appendix A.

5.2 ENVIRONMENT task data

Single-part placement instructions To test Task (i), we generated 20 simple, synthetic sentences containing a placement instruction of a single object, with all the needed parameters in the input. As explained in Section 3, the ENVIRONMENT parts are three-dimensional, may occupy one or two cells in the grid, and are gravity-bound. Since there are no prior placed objects in these instructions, gravity constraints mean that we only specify the $[x, y]$ coordinates in these sentences. The sentence structures are the following:

- Place a `[COLOR] [PART]` at the `[X]`th column, `[Y]`th row.
- Place a `[COLOR] [PART]` at the `[relative coordinate]` of the board.

For horizontal bridges, two columns are provided for the $[x_1, x_2]$ placement requirement, and, for vertical bridges, two rows for $[y_1, y_2]$. The relative coordinates may be grid positions such as *top left*, *middle*, etc., to be interpreted by the LLM agents. This data captures the two ways in which a user may fully define the placement of a part, either with absolute or relative coordinates.

Sequences of single-part instructions For Task (ii), the objective is to extract multiple objects from the same input. The input is composed of two sentences, with the same structure as the single instruction sentences. We created 13 tuples of these sentences with three possible variations: either the two parts are independently placed from each other, or the second part's placement is dependent on the first part's position (either on top or adjacent).

Construction of complex shapes Task (iii) asks for the construction of 10 complex shapes with sequences of instructions from an Architect. Each instruction includes the placement of one or more parts (standard or complex), with either absolute or relative placements (e.g., "*place a horizontal row of four purple gaskets*"). A list of the shapes is in Table 3; the shapes are constructed from 2 to 6 instructions. Some of these contain the same type of part, while others have a variety of parts and colors. The Moroccan Bridge structure is a highly complex shape composed of 18 ENVIRONMENT parts. A detailed list and a visualization of the shapes in the ENVIRONMENT environment is in Appendix G.1.

Underspecified instructions Task (iv) tests the handling of underspecified instructions; we split this task into underspecified single instructions (as in Task (i)) and underspecified two-part instructions (as in Task (ii)). For the former, we created 81 sentences, similar to those of Task (i), but with one piece of information missing: the part name (e.g. "it" instead of "nut"), the color, or one or all of the coordinates. For the latter, we created 202 instructions, in which either one or both parts described in the instruction were underspecified: the first single part placement was fully specified in 40 of those, and the second part placement was fully specified in 73 of the instructions.

Learning abstract functions in context Task (v) tests a model's ability to learn abstract functions given a context in which we specify an instance. We give instructions to create 9 complex structures, and we then ask the models to store them and recreate them in future instructions without any explicit information on their previously mentioned components. The first 4 structures are composed of 2-3 parts and have novel names (e.g., *A20 tower*) in order to ensure that the LLMs did not retrieve

shape information from representations learned in pretraining. The other 5 structures have common names (e.g., *square*, *skull*) but are composed of significantly more parts (16-18 for moderate complexity, up to 62 for complex structures). See Table 6 for a list of the names and Appendix G.2 for details.

6 Results on ENVIRONMENT tasks

Single-part placement instructions The results for Task (i) are in Table 1. We evaluated the accuracy of identifying the part type (e.g., *washer*, *hexagonal nut*), the color of the part, and the coordinates of the part on the grid. The baselines, the agentic system, and COCORELI, could all identify the type and colors. However, while the agentic LLM and COCORELI identified correct coordinates, the CoT single LLMs struggled with this.

Simple two-part instruction sequences In this task, we evaluate the accuracy of the first and second instructions separately, since each part has a separate object and property. The results are in Table 2. The second part of the sequence may have either independent coordinates ("Independent") or dependent coordinates ("Dependent") to an anchor part ("Dep. Anchor"). Our system performed well in this task and correctly identified the object properties for both objects in the input, as did the agentic model. The CoT LLMs were successful in extracting information for the first object of the input, but struggled to properly identify the second, with a 23-38% drop in accuracy.

Construction of complex shapes The results for the construction of complex shapes with a series of instructions are in Table 3. If a model/system has reached 100% accuracy, the complex structure has been recreated correctly since all the instructions were interpreted correctly step-by-step. The agentic model was successful in 7 out of 10 shapes, with CoT Claude-3.5 Sonnet being second with 6 out of 10 shapes, COCORELI recreating 5 out of 10 shapes, and CoT GPT-4.1 only 4 out of 10. However, looking at the number of instructions that the models were able to follow per complex structure, COCORELI was the only system capable of parsing (even partially) instructions for every shape, including the complex Moroccan bridge structure that none of the other models were able to handle. COCORELI demonstrates a 10% accuracy improvement in following instructions overall, compared to the single and agentic LLMs.

Accuracy	GPT-4.1	Claude	Agentic	COCORELI
Part type	**100**	**100**	**100**	**100**
Part color	**100**	**100**	**100**	**100**
Coordinates	81.67	76.67	**100**	**100**
TOTAL	93.89	92.23	**100**	**100**

Table 1: Fine-grained accuracy results (%) on Task (i): Single-part placement.

Accuracy	GPT-4.1		Claude		Agentic		CO..RELI	
	1st	2nd	1st	2nd	1st	2nd	1st	2nd
Part type	**100**	61.54	**100**	76.9	**100**	**100**	**100**	**100**
Part color	**100**	69.23	**100**	76.9	**100**	**100**	**100**	**100**
Coordinates	**100**	–	**100**	–	**100**	–	**100**	–
Independent	–	**100**	–	**100**	–	**100**	–	**100**
Dependent	–	63.64	–	69.23	–	**100**	–	**100**
Dep. Anchor	–	63.64	–	69.23	–	**100**	–	**100**
TOTAL	85.81		89.23		**100**		**100**	

Table 2: Fine-grained accuracy results for Task (ii): Simple two-part instruction sequences, for the first and second instructions of the sequence. Dashes indicate "not applicable".

Shape	GPT-4.1	Claude	Agentic	COCORELI
A	66.67	50.00	**100**	**100**
B	50.00	**100**	**100**	75.00
C	**100**	**100**	**100**	**100**
D	0	50.00	**100**	25.00
E	**100**	**100**	**100**	75.00
G	60.00	80.00	0	**100**
X	0	**100**	50.00	**100**
Square	**100**	**100**	**100**	75.00
+	**100**	**100**	**100**	**100**
Moroccan	0	0	0	57.1
TOTAL	54.67	69.04	69.04	**78.57**

Table 3: Accuracy (%) on following series of instructions in Task (iii): Construction of complex shapes.

Underspecified instructions The results for the subtask of evaluating underspecified single-part instructions are found in Table 4. As a first step, we identify whether the systems are able to detect that the input is missing information ("Missing info detected"), by returning the needed parameters as *null*. Most systems, including ours, successfully assigned *null*, except for Claude-3.5 Sonnet. The next step is to ask a clarification question targeting the missing information; only the agentic LLM and COCORELI could do this step. After the missing information is presented, both systems could append the additional information without mistakes

Shape	GPT-4.1	Claude	Agentic	COCORELI
Missing Info Detected	**100**	82.8	**100**	**100**
CQs Asked	N/A	N/A	**100**	**100**
Correct Parses (After CQ)	N/A	N/A	**100**	**100**
Hallucinations:				
on Part types	0.00	0.00	0.00	0.00
on Part colors	0.00	0.00	0.00	0.00
on Coordinates	0.00	17.2	0.00	0.00

Table 4: Accuracy (%) on Task (iv): underspecified instructions describing one part.

Shape	GPT-4.1	Claude	Agentic	COCORELI
Missing Info Detected	75.00	43.47	83.67	**100**
CQs Asked	N/A	N/A	83.67	**100**
Correct Parses (After CQ)	N/A	N/A	**100**	**100**
Hallucinations				
On Parts	32.69	38.46	34.21	0.00
On Colors	27.31	56.94	19.64	0.00
On Location	14.29	66.67	1.00	0.00

Table 5: Accuracy (%) on Task (iv): underspecified instructions describing two parts.

and avoid hallucinations. We also evaluate the tendency of the LLMs to hallucinate when required information is missing; we observe a low rate of hallucinations, except for Claude, which hallucinated on the part's coordinates on some occasions.

The results for the Task (iv) subtask, underspecified two-part instructions, are in Table 5. In this scenario, the instruction includes two parts, and either one or both are underspecified. Only COCORELI was completely successful on this subtask, correctly identifying missing input information, acquiring and parsing the needed information through the clarification loop, and avoiding hallucinations on properties. In contrast, the CoT single LLMs struggle to detect missing information in the input and hallucinate severely on all properties of the parts. The agentic model also strongly hallucinates on part properties but manages to identify some missing information, albeit less successfully than in the previous subtask. We also analyze the user burden arising from the underspecifications in Appendix E

Instructions for learning abstract functions in context In Table 6, we present the results of learning in context new functions for our 9 novel shapes. Only COCORELI was successful in the abstract representation and recreation of all shapes. We

Shape ID	GPT-4.1	Claude	Agentic	COCORELI
A20 tower	**T**	**T**	**T**	**T**
C15 stack	**T**	**T**	**T**	**T**
D21	**T**	**T**	**T**	**T**
X34	**T**	**T**	**T**	**T**
Square	**T**	**T**	**T**	**T**
Triad	**T**	**T**	F	**T**
Face	**T**	**T**	F	**T**
I	F	F	F	**T**
Skull	F	F	F	**T**

Table 6: Evaluation on Task (v): Shape reproducibility from instructions for learning functions in context

evaluate a reproduced shape R of an original shape O as correct if there is a bijection f between the basic parts X of R and basic parts of O, such that $\forall x \in X$ x have the same relative coordinates to the other parts x' of R as $f(x)$ has to $f(x')$ in O. All systems learned and recreated the simple, novel structures (*A20 tower*, *C15 stack*, *D21*, *X34*). The agentic model failed on medium-sized ones (16–18 objects). CoT LLMs succeeded on the medium shapes, but only COCORELI, however, solved all, including the most complicated shapes.

6.1 Discussion

The simple, single-part placement instructions, proved feasible for all systems. The task with two single-part instructions (Task ii), proved more challenging for the CoT baselines, whose performance on identifying the second object's properties was significantly lower. Task (iv) confirms this finding, with the models hallucinating to a greater extent on the properties of the underspecified second part. On the tasks of following a series of instructions (Tasks (iii) and (v)), the CoT models mainly were successful on simple structures, with less than 18 instructions overall. However, their failure cases (see Task (iii)) were severe, and their limited success on Task (v) relied on repeating previous inputs rather than utilizing external memory or abstraction abilities. Importantly, the CoT LLMs are *in principle* unable to resolve underspecification (Task iv), as they could not ask clarification questions. Agentic approaches perform significantly better in these tasks, as they can parse multiple objects, exploit user feedback, and retrieve objects more efficiently.

The agentic LLM baseline and COCORELI both excel in tasks for single instruction placement (Tasks (i) and (iv)), without the hallucinations of the CoT LLMs. Concerning Task (ii), the agentic LLM did better than our system in identify-

ing the two parts' properties; however, on more complex structures, such as in Tasks (iii) and (v), COCORELI outperformed the agentic LLM, by retrieving relational graphs, reducing the task to parsing, not reasoning. Regarding the Moroccan bridge structure in Task (iii), a very complex structure that involves learning a new shape, the agentic LLM failed to perform. Additionally, it was unable to recreate complex structures with more than 3 instructions in Task (v). We conclude that the agentic LLM has limited abilities compared to COCORELI, even with an LLM nine times larger than the one COCORELI uses.

Finally, concerning the tasks with underspecified instructions (Task (iv)), the superiority of the clarification loop approach was apparent. The COT systems were unable to resolve underspecifications and hallucinated. Meanwhile, the agentic LLM used the clarification loop when a single instruction was involved (Table 4) but showed significant problems when presented with a sequence of underspecified instructions. This did not pose a problem for COCORELI, which used the clarification loop well with a series of instructions.

7 Learning new functions in context in ToolBench

We conducted an additional experiment on in-context function learning as in Task v, but using the ToolBench data set (Qin et al., 2024). ToolBench is a dataset for evaluating unsupervised tool learning, featuring a library of functions that access APIs of various websites and their functionalities. As this task was primarily one of memory and not of task complexity, we compared COCORELI to our single-LLM CoT baselines (see Section 5.1).

Methodology After data preparation, which we detail in Appendix J, we provide: (1) the complete ToolBench workflow, with the user instructions and corresponding API function, and (2) new information. We then ask the system to call the API function with the new information. We made some adjustments to the COCORELI agent prompts. First, it parses the workflow instruction with the Instruction parser; then the Builder outputs a structured representation, `workflow_i`, along with the corresponding arguments (e.g., `arg1: address`, `arg2: message`). The Builder is then given a phrase, *this is a workflow*^{a} *_i*, which it learns to interpret as a command to abstract and store the abstracted `workflow`^{a}`_i` in the External memory. Thus, COCORELI can reapply `workflow`^{a} `_i` to new parameters. The single-LLM baseline CoT model (Claude 3.5 Sonnet) was prompted to return the API function with the new parameters. We evaluated only abstraction and generalization, as ToolBench lacks underspecified instructions and baselines lack a clarification mechanism

Metric	Precision	Recall	F1-Score
Claude	36.43	26.9	30.95
COCORELI	**100**	**100**	**100**

Table 7: Function reuse performance across models (GPT-4.1 omitted: 0 on all metrics).

Results on ToolBench The results of evaluating the generation and reapplication of 100 ToolBench function calls are in Table 7.[3] The single-LLM CoT baseline (Claude) struggled with the task of extracting a workflow function and applying it to new parameters, and completed the task for some of the data. COCORELI reapplied the new parameters without errors in ToolBench, indicating robustness beyond our ENVIRONMENT evaluation.

In both the ToolBench and ENVIRONMENT abstraction experiments, the relevant `workflows` or `concepts` are available in the immediate context. However, our method does not depend on this. Since the model's role is limited to parsing and emitting a structured representation (i.e., the workflow or concept name and its arguments), the use of abstraction and external memory naturally generalizes to scenarios where the workflow or concept was introduced earlier or taught previously.

8 Conclusion

COCORELI is a novel hybrid agentic approach using medium-sized LLM agents with a discourse module and external memory. It efficiently parses multiple user instructions, extracts object properties, and in-context learns abstract representations to reuse and adapt as needed. On the ENVIRONMENT tasks, COCORELI outperforms single-LLM CoT approaches and Agentic LLM-only systems, all of which use much larger LLMs: it avoids hallucinations, identifies missing information, asks the user for clarifications, and learns new objects. On ToolBench, COCORELI reused functions, whereas baselines only partially succeeded.

[3] Recall measures how many of the expected (gold) functions the model correctly reuses during output generation; precision gives the proportion of correct function uses to all function calls.

Limitations

We described a generic approach to agentic systems using abstract-apply functions that support tools through abstraction and application. While the concept is generalizable, its implementation must be tailored to each domain (e.g., ENVIRONMENT vs. ToolBench). Nevertheless, our current system requires that the API function definitions be formatted in JSON format. Real-world APIs don't come in this format, so this is an additional step to take into consideration.

Another limitation is that solving the ENVIRONMENT tasks ideally requires multi-modal capabilities. Our system handles only text. Our conversational component only handles one aspect of conversation. The detection of a missing feature in a function specification generates just one clarification question. We need to extend the conversational component to handle multiple *wh*-questions specifying various features at once. We also need to extend the system to handle other discourse moves, such as corrections, and longer question-answer sessions. A complete system for human-AI collaborative tasks would also need a robust planning system for handling high-level instructions.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. 2022. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. *Preprint*, arXiv:2204.01691.

Anthropic. 2024. Claude 3.5 Sonnet. [Accessed on 14.05.2025].

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. 2024. A survey on in-context learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 1107–1128, Miami, Florida, USA. Association for Computational Linguistics.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.

Luca Gioacchini, Giuseppe Siracusano, Davide Sanvito, Kiril Gashteovski, David Friede, Roberto Bifulco, and Carolin Lawrence. 2024. AgentQuest: A Modular Benchmark Framework to Measure Progress and Improve LLM Agents. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 3: System Demonstrations)*, pages 185–193, Mexico City, Mexico. Association for Computational Linguistics.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji,

Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. 2024. The Llama 3

Herd of Models. *Preprint*, arXiv:2407.21783.

W. Jenkins, B. Mather, and D. Munson. 1985. Nearest neighbor and generalized inverse distance interpolation for fourier domain image reconstruction. In *ICASSP '85. IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 10, pages 1069–1072.

Ingmar Kanitscheider, Joost Huizinga, David Farhi, William Hebgen Guss, Brandon Houghton, Raul Sampedro, Peter Zhokhov, Bowen Baker, Adrien Ecoffet, Jie Tang, Oleg Klimov, and Jeff Clune. 2021. Multi-task curriculum learning in a complex, visual, hard-exploration domain: Minecraft. *Preprint*, arXiv:2106.14876.

Yi Kong, Dianxi Shi, Guoli Yang, Zhang ke di, Chenlin Huang, Xiaopeng Li, and Songchang Jin. 2025. Mapagent: Trajectory-constructed memory-augmented planning for mobile task automation. *Preprint*, arXiv:2507.21953.

Bill Yuchen Lin, Yicheng Fu, Karina Yang, Faeze Brahman, Shiyu Huang, Chandra Bhagavatula, Prithviraj Ammanabrolu, Yejin Choi, and Xiang Ren. 2023. Swiftsage: A generative agent with fast and slow thinking for complex interactive tasks. *Preprint*, arXiv:2305.17390.

Zichuan Lin, Junyou Li, Jianing Shi, Deheng Ye, Qiang Fu, and Wei Yang. 2022. JueWu-MC: Playing Minecraft with Sample-efficient Hierarchical Reinforcement Learning. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 3257–3263. International Joint Conferences on Artificial Intelligence Organization. Main Track.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. 2024. Agentbench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*.

Jieyi Long. 2023. Large language model guided tree-of-thought. *Preprint*, arXiv:2305.08291.

Nikhil Mehta, Milagro Teruel, Patricio Figueroa Sanz, Xin Deng, Ahmed Hassan Awadallah, and Julia Kiseleva. 2024. Improving grounded language understanding in a collaborative environment by interacting with agents through help feedback. *Preprint*, arXiv:2304.10750.

Seyed Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2025. GSM-symbolic: Understanding the limitations of mathematical reasoning in large language models. In *The Thirteenth International Conference on Learning Representations*.

Omar Naim and Nicholas Asher. 2025. Two in context learning tasks with complex functions. *Preprint*, arXiv:2502.03503.

Omar Naim, Guilhem Fouilhé, and Nicholas Asher. 2024. Re-examining learning linear functions in context. *Preprint*, arXiv:2411.11465.

Anjali Narayan-Chen, Prashant Jayannavar, and Julia Hockenmaier. 2019. Collaborative Dialogue in Minecraft. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5405–5415, Florence, Italy. Association for Computational Linguistics.

OpenAI. 2025. Introducing GPT-4.1 in the API. [Accessed on 14.05.2025].

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. Art: Automatic multi-step reasoning and tool-use for large language models. *Preprint*, arXiv:2303.09014.

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. Talm: Tool augmented language models. *Preprint*, arXiv:2205.12255.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, New York, NY, USA. Association for Computing Machinery.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large language model connected with massive apis. In *Advances in Neural Information Processing Systems*, volume 37, pages 126544–126565. Curran Associates, Inc.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah Smith, and Mike Lewis. 2023. Measuring and Narrowing the Compositionality Gap in Language Models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5687–5711, Singapore. Association for Computational Linguistics.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*.

Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Garð ar Ingvarsson, Timon Willi, Ravi Hammond, Akbir Khan, Christian Schroeder de Witt, Alexandra Souly, Saptarashmi Bandyopadhyay, Mikayel Samvelyan, Minqi Jiang, Robert Lange, Shimon Whiteson, Bruno Lacerda, Nick Hawes, Tim Rocktäschel, Chris Lu,

and Jakob Foerster. 2024. JaxMARL: Multi-Agent RL Environments and Algorithms in JAX. In *Advances in Neural Information Processing Systems*, volume 37, pages 50925–50951. Curran Associates, Inc.

Gabriel Sarch, Yue Wu, Michael J. Tarr, and Katerina Fragkiadaki. 2023. Open-ended instructable embodied agents with memory-augmented large language models. *Preprint*, arXiv:2310.15127.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, pages 68539–68551. Curran Associates, Inc.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Preprint*, arXiv:2303.11366.

Malavikha Sudarshan, Sophie Shih, Estella Yee, Alina Yang, John Zou, Cathy Chen, Quan Zhou, Leon Chen, Chinmay Singhal, and George Shih. 2024. Agentic LLM Workflows for Generating Patient-Friendly Medical Reports. *Preprint*, arXiv:2408.01112.

Chris Sypherd and Vaishak Belle. 2024. Practical Considerations for Agentic LLM Systems. *Preprint*, arXiv:2412.04093.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *Preprint*, arXiv:2306.05301.

Jen tse Huang, Eric John Li, Man Ho Lam, Tian Liang, Wenxuan Wang, Youliang Yuan, Wenxiang Jiao, Xing Wang, Zhaopeng Tu, and Michael R. Lyu. 2025. How Far Are We on the Decision-Making of LLMs? Evaluating LLMs' Gaming Ability in Multi-Agent Environments. *Preprint*, arXiv:2403.11807.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024. Voyager: An Open-Ended Embodied Agent with Large Language Models. *Transactions on Machine Learning Research*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. *Preprint*, arXiv:2203.11171.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

Yue Wu, Xuan Tang, Tom Mitchell, and Yuanzhi Li. 2024. Smartplay : A benchmark for LLMs as intelligent agents. In *The Twelfth International Conference on Learning Representations*.

Lin Xu, Zhiyuan Hu, Daquan Zhou, Hongyu Ren, Zhen Dong, Kurt Keutzer, See-Kiong Ng, and Jiashi Feng. 2024. MAgIC: Investigation of large language model powered multi-agent in cognition, adaptability, rationality and collaboration. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 7315–7332, Miami, Florida, USA. Association for Computational Linguistics.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems*, volume 36, pages 11809–11822. Curran Associates, Inc.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023b. ReAct: Synergizing Reasoning and Acting in Language Models. *Preprint*, arXiv:2210.03629.

Honghua Zhang, Liunian Harold Li, Tao Meng, Kai-Wei Chang, and Guy Van den Broeck. 2023. On the paradox of learning to reason from data. In *IJCAI*, pages 3365–3373.

Yufan Zhuang, Xiaodong Yu, Jialian Wu, Ximeng Sun, Ze Wang, Jiang Liu, Yusheng Su, Jingbo Shang, Zicheng Liu, and Emad Barsoum. 2025. Self-taught agentic long context understanding. *Preprint*, arXiv:2502.15920.

Appendices

A Technical specifications

Baseline single LLM models from commercial APIs:

1. Claude-3.5-Sonnet (175 billion parameters) API from the official API: https://claude.ai/new
2. GPT-4.1 (unknown parameter size) from the official API: https://platform.openai.com/docs/models/gpt-4.1

For the Agentic Model, we used Llama-3.3-70b-Instruct, through the AI/ML API service: https://aimlapi.com/app/.

For the design and implementation of COCORELI, we used LLaMA-3.1 8b from the Ollama library (https://ollama.com), and from the `transformers` library from HuggingFace (https://huggingface.co/). All 8b experiments were done locally without using any commercial APIs. To run the Hugging Face models locally, we used Volta GPUs from NVIDIA.

B Ablation Study

B.1 Controlled and implicit ablations

We first performed small **controlled ablations** by disabling single modules on the task families where they are structurally required. As shown in Table 8, each removal leads to categorical collapse (0% completion or 100% failure), indicating that these components are essential for system functionality.

Ablation	Task subset (N)	Outcome
Clarification loop disabled	20	0 / 20 completions
External memory disabled	5	0 / 5 abstractions
Parser/Builder/Locator sep. removed	10	10 / 10 invalid JSONs or crashes

Table 8: Controlled ablation results on representative subsets of the relevant task families. Each removal produces categorical collapse (0% completion or 100% failure), confirming that these modules are structurally necessary.

While these controlled removals establish that modules are indispensable, they do not reveal *how* each contributes. Because COCORELI is prompt-driven, destructive ablations would require rewriting prompts, effectively creating a new system rather than a controlled variant. We therefore report **task-based ablations**, where distinct evaluation regimes (simple parsing, multi-part planning, ambiguous inputs, and abstraction with memory) naturally isolate the role of each module without altering the architecture.

- **Parsing only (Tasks i–ii).** With 1–2 parts, COCORELI achieves 100% accuracy, isolating pure parsing ability (Table 1, Table 2).
- **Complex multi-part instructions (Task iii).** Accuracy drops as implicit planning load increases, highlighting the limits of planning without a dedicated planner (Table 3).
- **Ambiguous instructions (Task iv-a,b).** Underspecified fields produce `null` in the JSON, which deterministically triggers a clarification question (CQ). This shows the loop's necessity: without it, the system hallucinates or stalls (Table 4, Table 5).
- **Abstraction and reuse (Task v).** Once parsing succeeds, retrieval from external memory is deterministic, so COCORELI reproduces complex shapes with 100% accuracy. Without memory, no reuse is possible (0%), showing that memory is essential for enabling abstraction and reuse (Table 6).

Together, the controlled and task-based ablations show both the necessity and the functional role of parsing, clarification, and memory, without requiring ad-hoc prompt rewrites.

C More Related Work

Prompting techniques LLMs falter at generalization, since it requires abstraction beyond the scope of their learned representations (Zhang et al., 2023). Approaches such as *in-context learning* (ICL) provide additional information to the LLM in the form of an analogy, thus expanding their domain knowledge, for example, in mathematical equations (Naim et al., 2024; Mirzadeh et al., 2025; Naim and Asher, 2025). However, ICL remains limited to domain adaptation, lacking generalizability (Dong et al., 2024).

Chain-of-thought (CoT) prompting (Wei et al., 2022; Wang et al., 2023) aims to improve performance by asking the LLM to resolve a task by documenting its solution step by step. SayCan by Ahn et al. (2022) supply an LLM with the list of simpler tasks and prompt the model to decompose a complicated task as a sequence of simpler tasks. Press et al. (2023) introduce multi-hop questions in CoT prompting with answers that require multiple facts that are novel to the LLM. The ReACT

framework by Yao et al. (2023b) extend the CoT process by integrating information retrieval from the Wikipedia API. Yao et al. (2023a) introduce trees of thought (ToT) in order to diversify the LLM's decision-making process to improve performance. Long (2023) also enhance CoT with ToT, and incorporate agents and external memories.

Agentic models Several studies have tested single-agent LLM models on their abilities in complex interactive tasks, including games, and report weaknesses in reasoning and planning (Liu et al., 2024; Gioacchini et al., 2024; Wu et al., 2024). Specifically on Minecraft, Wu et al. (2024) report very low performance of multiple LLMs compared to human competencies and problems, such as with 3D spatial reasoning. Larger LLMs (GPT3.5, LLaMA-3.1 70B) still outperform smaller, novel LLMs in game benchmarks (tse Huang et al., 2025). Regarding multi-agent approaches, Park et al. (2023) introduced Generative Agents, a system based on GPT-3.5 that uses independent agents, each capable of utilizing functions for memory retrieval, reflection generation, and sequence planning. Rutherford et al. (2024) used multi-agent reinforcement learning (MARL), testing in a variety of tasks in experimental computational game theory. Xu et al. (2024) assigned cognition, adaptability, rationality, and collaboration objectives to multiple agents, using probabilistic graphic modeling. They reported that the larger LLMs, such as GPT -4 Turbo, had the best performance on game benchmarks.

Interactive and memory-augmented agents Mehta et al. (2024) study interaction in Minecraft-style environments, where agents can receive help feedback from humans or simulated sources. Their approach is reactive and correction-oriented: the agent acts, and when errors occur, feedback is integrated to recover. In contrast, the ENVIRONMENT benchmark assumes that wrong placements are not permissible, making proactive clarification essential. COCORELI's discourse module therefore resolves underspecification before execution, whereas Mehta's framework is not directly applicable under ENVIRONMENT's stricter setting.

Sarch et al. (2023) introduce HELPER, an embodied agent with a memory-augmented LLM that retrieves past language–program pairs to improve generalization in instruction following. While effective on TEACh, HELPER's reliance on retrieval limits its ability to handle novel underspecified tasks, as it lacks the compositional abstractions (e.g., coordinates, parts, colors) needed for recomposition in ENVIRONMENT. Moreover, HELPER's feedback mechanism is reactive and post-execution, asking for corrections after acting, whereas ENVIRONMENT requires proactive clarification before any placement, since errors are not permissible. Consequently, HELPER's memory and correction style are not directly applicable to the structured underspecification benchmarks that COCORELI addresses.

Tool learning Tool learning involves the use of external tools to improve problem solving and broaden the functional scope of the LLM, thus augmenting performance on specific downstream tasks (Parisi et al., 2022; Gao et al., 2023; Long, 2023). Recently, researchers have developed self-supervised tools that can support a level of abstraction and generalization, making them useful across multiple applications. Schick et al. (2023) developed Toolformer, an LLM with GPT-3 in its underlying architecture, that is capable of generating functions to use API tools from various websites. ToolLLM (Qin et al., 2024) has a similar motivation in generating API calling functions with LLaMA-2 7B in an unsupervised manner, in addition to proposing ToolBench, a dataset for evaluating unsupervised tool learning. Gorilla (Patil et al., 2024) is also an API function generating tool that incorporates document retrieval from the target API documentation. Tang et al. (2023) introduced ToolAlpaca, a framework to fine-tune LLMs for better tool learning of API functions.

D Cost Analysis

Table 9 compares efficiency across different setups for instruction-to-action parsing.

The **CoT systems** (GPT-4.1, Claude) incur direct API costs per query. Their outputs are compact (around 20–40 tokens), which keeps costs modest, but differences in vendor pricing lead to GPT averaging around $6 per 1K queries and Claude around $10. Latency is flat (8–11s) across specified and underspecified tasks because these models do not attempt clarification—underspecified inputs are simply passed through with `null` placeholders. This avoids additional cost but leaves instructions incomplete.

The **Agentic baseline** (LLaMA-70B) uses multi-step reasoning with longer outputs (around 150–300 tokens). When underspecified, it may is-

sue clarification questions, which slightly increases output length and cost, but this behavior is inconsistent: some queries are left unresolved while others expand into long reasoning traces. As a result, the underspecification penalty manifests as stochastic increases in token usage, with the burden of resolution falling on the model's exploratory generation.

COCORELI (LLaMA-8B) runs locally and therefore incurs *no* API costs. Its outputs are structured JSON parses, where missing fields are explicitly marked as `null`. A deterministic Python layer **Executor module** then converts these into targeted clarification questions. This design shifts underspecification handling out of the LLM itself: the model cost remains constant across specified and underspecified tasks, and clarifications are resolved predictably without bloating output sequences.

In short, GPT and Claude bypass underspecification, Agentic absorbs it stochastically in model outputs, and COCORELI handles it deterministically outside the model, trading modest latency for zero API cost and predictable clarification.

We do not report efficiency for the abstraction experiments Task 6, since these are one-shot queries rather than repeated instruction following. Nonetheless, the scaling behavior is informative: for **CoT systems**, output size grows directly with the number of parts (e.g., regenerating 60 `place(...)` calls is much more expensive than 6). For the **Agentic baseline**, outputs scale with the complexity of the induced program (longer Python code for more complex structures). By contrast, **COCORELI** abstracts structures into a fixed-size JSON schema, so output size is independent of the structure's size or complexity. This makes COCORELI's abstraction mechanism cost-invariant, even as structures grow large.

System	Task	Tokens	Latency(s)	Cost($/1K)
GPT	1S	2818/17 [2835]	10	5.77
	2S	2829/33 [2862]	10	5.92
	1U	2816/17 [2833]	11	5.77
	2U	2829/33 [2862]	10	5.92
Claude	1S	3096/19 [3115]	8	9.57
	2S	3139/38 [3177]	8	9.99
	1U	3126/19 [3145]	8	9.66
	2U	3139/38 [3177]	8	9.99
Agentic	1S	2823/150 [2973]	9	2.79
	2S	2835/299 [3134]	9	2.94
	1U	2823/150 [2973]	9	2.79
	2U	2835/299 [3134]	9	2.94
COCORELI	1S	4623/141 [4764]	12	0.00
	2S	4635/282 [4917]	12	0.00
	1U	4623/141 [4764]	10	0.00
	2U	4635/282 [4917]	12	0.00

Table 9: Efficiency metrics across systems and task types. Tokens are average per-instruction values, reported as input/output [total]. Costs are reported in U.S. dollars per 1,000 queries, computed from official per-million input/output token pricing (CORELI runs locally, cost 0.00). 1S = one-part specified, 2S = two-part specified, 1U = one-part underspecified, 2U = two-part underspecified. Task 3 omitted since costs scale proportionally with instruction length while relative ordering is preserved (accuracy in Section 3).

E User Burden Analysis

To evaluate the practical impact of our system on end users, we conduct a *user burden study* that quantifies both the correctness and the effort required during interaction. In the context of our CoCoBots setup, user burden primarily arises from the clarifying questions (CQs) that the system issues to resolve underspecified instructions.

E.1 Metrics for Underspecified Instructions

For instructions that are partially underspecified, we define the following metrics with respect to the gold standard set of missing information:

- **True Positive (TP) CQs**: A CQ that correctly targets missing information. This reflects *necessary questions* that genuinely reduce ambiguity for the user.
- **False Positive (FP) CQs**: A CQ that asks for information which is already fully specified. These represent *unnecessary interruptions*, increasing cognitive load and interaction time.
- **False Negative (FN) CQs**: Missing information that the system fails to ask about. FN CQs indicate situations where the user must provide additional guidance on their own.
- **Precision, Recall, F1**: Derived from TP, FP, and FN. Precision reflects the fraction of CQs that were necessary (low FP), recall measures the fraction of missing information addressed (low FN), and F1 captures the balance. These metrics quantify *user burden in terms of correctness*.

By combining correctness (or user-burden) metrics (TP, FP, FN, precision, recall, F1 for underspecified instructions) with unnecessary CQs (for fully specified instructions) and efficiency measures (tokens, time, cost), we provide a comprehensive and objective assessment of user burden.

These metrics provide a reproducible proxy for user burden. While actual human perception may vary, unnecessary and missed clarifications, as well as time and token usage, directly reflect potential

System	Gold	TP	FP	FN	Prec.	Rec.	F1
COCORELI	128	128	0	0	1	1	1
Agentic	128	110	18	0	85.9	1	92.4

Table 10: User Burden Metrics for 1 part Fully Underspecified Instructions (single part). TP = True Positives (correctly asked CQs / coverage of missing info), FP = False Positives (unnecessary CQs asked for already specified attributes), FN = False Negatives (missed CQs).

System	Gold	TP	FP	FN	Prec.	Rec.	F1
COCORELI	567	567	0	0	1.00	1.00	1.00
Agentic	567	393	81	93	82.9	80.9	81.9

Table 11: Clarification question (CQ) accuracy for 2-part underspecified tasks. TP = true positives, FP = false positives, FN = false negatives.

cognitive and operational load on the user.

F COCORELI & Baseline Prompts

F.1 Single-LLM CoT Prompts

"You are an expert at translating natural language instructions into specific actions within a 3D grid environment.
∗∗Environment Information∗∗
The environment is a 16x16x16 3D grid where each cell represents a single block. Rows run along the X-axis and columns along the Y-axis (both from 1 to 16), while the Z-axis (from 1 to 16) indicates height.
The ground level is at Z=1, and higher Z-values represent increased elevation.
Available parts: ["screw", "nut", "washer", "horizontal bridge", "vertical bridge", "bolt", "gasket", "hex nut", "square nut"] .
Available colors: ["blue", "orange", "red", "green", "yellow", "purple", "black", "white", "brown", "magenta"] .
∗∗Task:∗∗
You are given a natural language instruction from the user and a 3D grid environment. Your task is to translate the instruction into a series of actions that the AI system can perform to construct and modify the structure of the environment.
First, you need to understand the instruction and determine if it is valid or invalid. If it is invalid, provide a clarifying question in a natural manner.
For valid instruction, return a list of actions to be executed on the environment.
Only include actions that call the available functions (e.g., `place()` or `remove()`)." If the plan is purely informational or does not require actions, return an empty list.
Make sure to return only in this format: place(part, color, row, column, height) or remove(part, color, row, column, height)"

F.2 Agentic Prompts

F.2.1 Environment

" ∗∗Environment Information∗∗
The environment is a $16 \times 16 \times 16$ grid where each cell represents a single block. The grid uses a standard XYZ coordinate system:

- The ∗∗X-axis∗∗ (values 1 to 16) runs horizontally (left-to-right) and defines the grid's columns.
- The ∗∗Y-axis∗∗ (values 1 to 16) runs vertically on each horizontal layer (top-to-bottom when viewed as an XY plane) and defines the grid's rows.
- The ∗∗Z-axis∗∗ (values 1 to 16) represents vertical elevation, with Z=1 as ground level and higher Z-values indicating greater elevation.

Each horizontal layer at a fixed Z value is an XY plane. Within any given XY plane:

- A ∗∗vertical row∗∗ is the sequence of blocks where the X-coordinate remains constant while the Y-coordinate varies. This forms a column (when viewed on the XY plane) that runs from the top to the bottom of that plane.
- Similarly, a horizontal row is where the Y-coordinate remains constant and the X-coordinate varies.

A column is a succession of parts, having successive 'y' coordinates and the same 'x' and 'z' coordinates.
A row is a succession of parts, having successive x coordinates and the same 'y' and 'z' coordinates.
The first row is the row at the top, from the upper view. And the first column is the column on the left by the upper view.
Available parts: ["screw", "nut", "washer", "horizontal bridge", "vertical bridge", "bolt", "gasket", "hex nut", "square nut"] .
Available colors: ["blue", "orange", "red", "green", "yellow", "purple", "black", "white", "brown", "magenta"] . "

F.2.2 Instruction Parser

"You are an expert at extracting and designing structural plans from natural language instructions in a 3D grid environment.
(Environment)
∗∗Task:∗∗

Interpret the provided instruction and determine the structural designs implied. Your response should address two scenarios:
1. **Known Structures: **
- If the instruction refers to structures that correspond to existing tools (such as those for placing or removing parts), provide a brief explanation of how these actions will be performed. Include relevant details (e.g., part type, color, and coordinates) in plain language.

2. **New Structures: **
- If the instruction implies a new or custom structure not covered by existing tools, generate a clear, descriptive plan that includes:
- A concise description of the new structure.
- A list of key parameters required to define the structure (for example, dimensions, orientation, and position).
- A proposed name for the structure if one is not provided in the instruction.

Your output should first provide a brief natural language explanation of how you interpret the instruction. Then, for any new structures identified, include a detailed plan as described above.
Do not call any functions or include code in your response.

Output Format
MAKE SURE YOU OUTPUT VALID JSON. No text before or after JSON, no trailing commas, no comments (//), no unnecessary quotes, etc.
Output only the structure in the specified JSON format with the following keys without any additional information:
- `structures`(list of `StructurePlan`): A list of structures identified in the instruction. Each structure has the following fields:
  - `plan`(string): The plan for the structure to be executed by the agent.
  - `name`(string): The name of the structure.

F.2.3 Locator

"You are an expert at converting relative spatial references to absolute coordinates in a 3D grid.
(Environment)
Task:
Analyze the provided instruction and update it with precise, absolute locations for each referenced part. If the instruction already contains absolute positions, return it unchanged. Do not call any functions in your response.
Output Format
MAKE SURE YOU OUTPUT VALID JSON. No text before or after JSON, no trailing commas, no comments (//), no unnecessary quotes, etc.
Output only the structure in the specified JSON format with the following keys without any additional information:
- `instruction`(string): The updated instruction with absolute positions.

F.2.4 Abstractor

"You are an expert Python developer specializing in pattern recognition and code generation.
(Environment)
Task:
Analyze the sequence of place actions and generate a Python function that can recreate the structure. The function should:
1. Identify patterns in the placement sequence
2. Group similar placements together
3. Use loops and helper functions to minimize code duplication
4. Make the code reusable and parameterized

Input Format:
You will receive a sequence of place actions in the format:
`Place a [color] [part] at row [y] column [x] height [z]`

Output Format:
Generate a Python function that recreates the structure. The function should:
- Accept parameters for customization (color, position, etc.)
- Use helper functions for common patterns
- Include clear documentation
- Return a list of place actions

Example Output Format:

```python
def build_structure(color: str, start_x:
```

int, start_y: int, start_z: int):
"Builds a structure with the given parameters."

Args:
color (str): Color of the parts
start_x (int): Starting x coordinate
start_y (int): Starting y coordinate
start_z (int): Starting z coordinate

Returns:
list: List of place actions
"""
actions = []

return actions
```

Make sure to analyze the input sequence carefully and identify any patterns or symmetries that can be used to simplify the code."

# G ENVIRONMENT dataset details

## G.1 Task (iii)

In these examples, every single instruction is fully specified. The instructions themselves are complex, drawing away from the simple template style approach we were following. Here is a sample:

> **[Turn 1]**<Architect>Starting on the second row and second column, place a magenta washer
> **[Turn 2]**<Architect> On the third row, a column of four more magenta washers. All in the second column
> **...**

A detailed list of the 10 complex shapes used for Task (iii): Construction of complex shapes

- **A**: 14 parts (magenta washers) | 6 instructions
- **B**: 10 parts (various parts/colors) | 4 instructions (see Figure 3)
- **C**: 8 parts (green screws) | 3 instructions
- **D**: 10 parts (various parts/colors) | 4 instructions
- **E**: 10 parts (various parts/colors) | 4 instructions
- **G**: 17 parts (various parts/colors) | 5 instructions
- **+**: 9 parts (magenta bolts) | 3 instructions
- **Square**: 16 parts (various bolts) | 4 instructions
- **X**: 10 parts (various bolts) | 2 instructions
- **Moroccan Bridge structure**: A composite shape, made up of a complex shape that has to be learned as a **Moroccan Bridge** (5 parts of various colors) and reused, alongside 5 other standard parts | 8 instructions (see Figure 4).

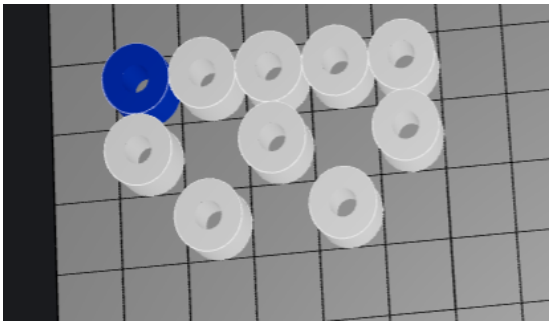


Figure 3: The B Shape

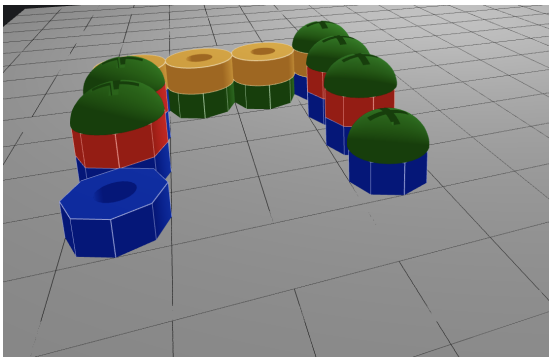


Figure 4: The Moroccan Bridge

## G.2 Task (v)

A **sample** of human input to COCORELI instruction from the input for the C15 shape:

- INST: Can you place a blue screw at row 4 column 5 height 1
- A: Place a blue screw at row 4 column 5 height 1
- INST: Place a red screw next to the blue screw, and put a red screw on top.
- A: Place a red screw at row 4 column 6 height 1
- A: Place a red screw at row 4 column 6 height 2
- INST: This is what I call a C15
- INST: Now make me another C15 at the eighth row and ninth column

We present a detailed list of the 9 complex shapes used for Task (v): Abstractive instructions. The instructions for these shapes are stored in the Dialogue History.
```

- **A20 tower**: 4 parts (various) | 5 instructions
- **C15 stack**: 7 parts (various) | 6 instructions (see Figure 5)
- **D21**: 6 parts (various) | 5 instructions
- **X34**: 5 parts (various) | 5 instructions
- **Square**: 16 parts (blue washers) | 17 instructions
- **Triad**: 18 parts (red nuts) | 17 instructions
- **Face**: 47 parts (blue nuts)
- **I**: 18 parts (blue nuts) (see Figure 9)
- **Skull**: 62 parts (blue nuts) | 63 instructions (see Figure 7)

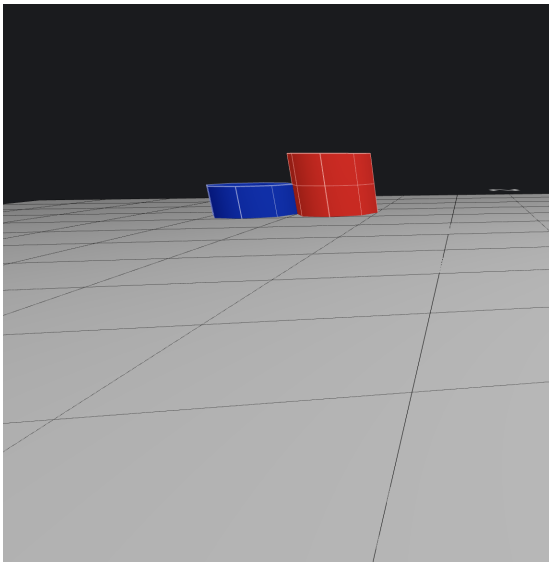


Figure 5: The C15 shape

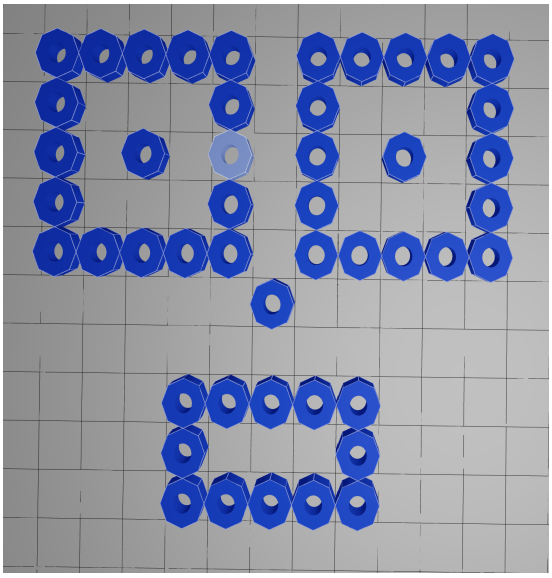


Figure 6: The Face shape

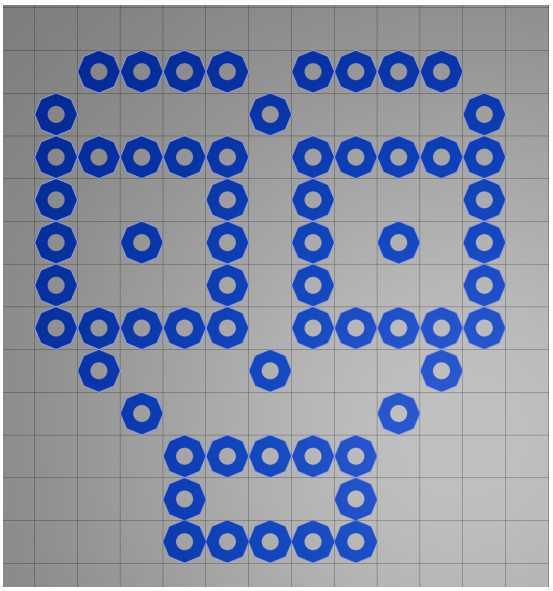


Figure 7: The Skull shape

H Failure cases by the models

We present two failure cases on Task (v) by the GPT-4.1 baseline. As discussed in Section 6 and shown in Table 6, the baseline was not able to reproduce the more complex shapes of the task from the dialogue history, but COCORELI was, through abstraction.

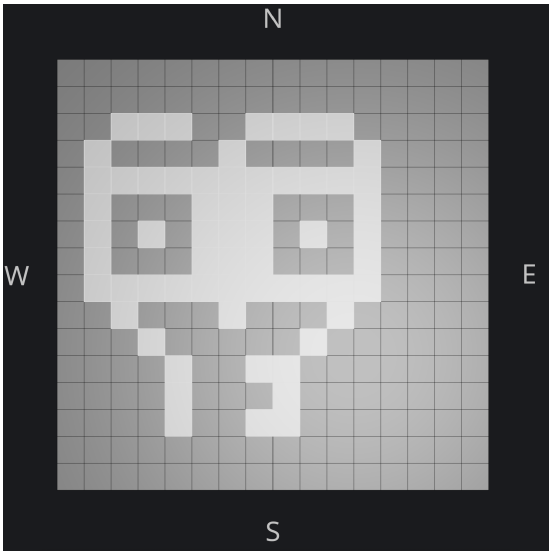


Figure 8: The skull that was recalled by GPT-4.1. The target shape can be seen in Figure 7.

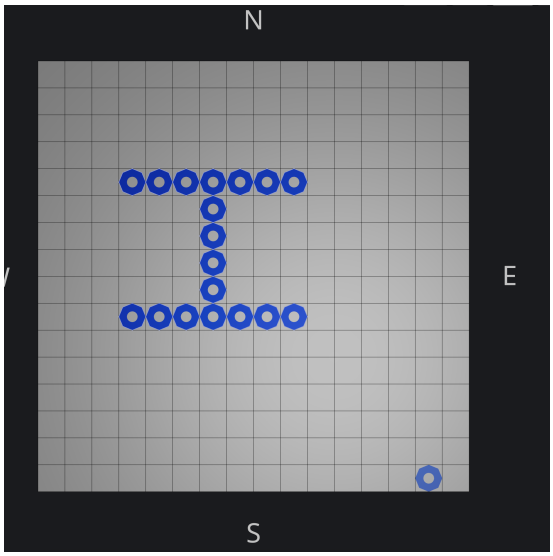


Figure 9: The original I shape for Task (v).

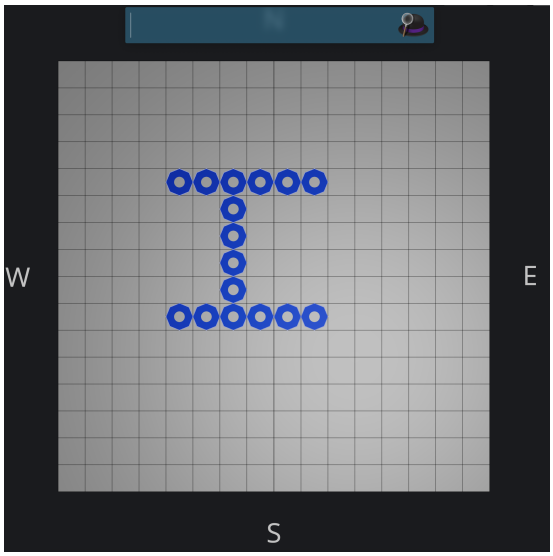


Figure 10: The I shape that was recalled by GPT-4.1 for Task (v).

I Algorithms for Abstraction and Reconstruction

Algorithm 1 Convert Structure to Graph to Abstract Coordinates

Require: S ▷ **S**:Structure containing exact coordinates and part names
C ← Number of connected parts in S
A ← 1 ▷ Initialise to the first connected component
G ← [ ] ▷ Empty Adjacency List for a single component
T ← { } ▷ Empty Dict to store all Adjacency Lists
while $C \neq 0$ **do** ▷ Iterate through all connected parts
  while v in A **do** ▷ Iterate through the vertices in A
    X ← parts in all adjacent coordinates of v
    G[v] ← X ▷ Vertex v stores information about all the 'neighbors'
    v ← v + 1 ▷ Move to the next vertex
  end while
  $T[C]$ ← G ▷ Add G to the final graph
  $G \leftarrow 0$ ▷ Flush G to store next component
  $C \leftarrow C - 1$ ▷ Move to the next component
end while

Algorithm 2 Compute Offsets from Reference Point

Require: S_t ▷ The user start coordinates for the first component
Require: T ▷ The Stored Graph for a structure
Firsts ← First vertices for each components
Ref ← Firsts[0] ▷ Store the first point as reference
offsets ← [] ▷ Initialize empty list to store offsets
Starts ← [] ▷ Initialize empty list to store start points
for each (x, y, z) in Firsts **do**
  Append $(x - Ref.x, y - Ref.y, z - Ref.z)$ to offsets
end for
for each (dx, dy, dz) in offsets **do**
  Append $(S_t.X + dx, S_t.Y + dy, S_t.Z + dz)$ to Starts
end for
return Starts ▷ Contains the new starts for each component

Algorithm 4 uses the nearest neighbor scaling algorithm (Jenkins et al., 1985), so it includes the limitations that are native to the main algorithm.

J Data preparation for Toolbench

Data We first extracted 100 workflows from the dataset, i.e., a sequence of user instructions, actions, and one or more API functions to fulfill the user's request. We ensured that they include a descriptive user instruction and an associated API function with at least two variables (non-boolean), in order to evaluate the abstractive abilities of our systems. For example, the user input "*Schedule a meeting with Alex tomorrow at 3 pm and send a confirmation email.*" leads to the functions `create_event(q, time)` and `send_email(q, message)`. These functions are abstract and can be reused with different variables.

Algorithm 3 Apply a stored structure at New user Location

Require: Starts ▷ The start coordinates to replicate a structure
Require: T ▷ The graph stored for the the required structure
C ← The list of connected components in T
C_{Starts} ← The start coordinates for each coordinate from (2) i ← 0 ▷ initialize to first component index
while i < len(C) **do** ▷ Iterate through the components
s ← C_{Starts}[i] ▷ Take the first component's start
BFS(C[i],s) ▷ Do a bfs traversal starting from s to generate co-ords respective to the start s
i ← i + 1
end while

Algorithm 4 Scale a shape to certain coordinates

Require: G ▷ The concrete coordinates as they are placed on the board
Require: T_x, T_y, T_z ▷ The target coordinates to scale to
S_x, S_y, S_z ← The bounding box for G
S_G ← Nearest_Neighbor([S_x, S_y, S_z],[T_x, T_y, T_z])
return S_G ▷ Return the new coordinates after scaling to the larger grid

Algorithm 5 Abstract2

Require: T ▷ The relative coordinates from (1)
▷ These values below come from the Locator and Chatter agents
C ← Y/N ▷ Yes/No whether to abstract color
P ← Y/N ▷ Yes/No whether to abstract parts
S ← Y/N ▷ Yes/No whether to abstract shape
return C,P,S ▷ Which parameters of a graph to abstract over

Algorithm 6 Apply2

Require: G ▷ Applied Graph from (3)
Require: C,P,S ▷ Output from (5) for graph G
if $C \leftarrow Y$ and $P \leftarrow Y$ and $S \leftarrow Y$ **then**
Require: C_{val} ← From Chatter(new color)
Require: P_{val} ← From Chatter and Locator(new Part)
Require: S_{val} ← From Chatter(new Dimension)
G' ← Algo(3) with C_{val}, P_{val}
Final ← Algo(4) on G' with S_{val}
end if
if $C \leftarrow N$ and $P \leftarrow Y$ and $S \leftarrow Y$ **then**
Require: P_{val} ← From Chatter and Locator(new Part)
Require: S_{val} ← From Chatter(new Dimension)
G' ← Algo(3) with P_{val}
Final ← Algo(4) on G' with S_{val}
end if
if $C \leftarrow N$ and $P \leftarrow N$ and $S \leftarrow Y$ **then**
Require: S_{val} ← From Chatter(new Dimension)
G' ← Algo(3) with P_{val}
Final ← Algo(4) on G' with S_{val}
end if

A sample datapoint looks like the following example :-

> **USER:** Schedule a meeting with Alex tomorrow at 3 pm and send a confirmation email.
> **ACTIONS:** `create_event(q: “Alex”, time: “2025-05-09 15:00”)`
> **ACTIONS:** `send_email(q: “Alex”, message: “Meeting Confirmation”, “See you at 3 pm tomorrow!”)`
> **USER:**This is `workflow_1`.
> **USER:**Please apply `workflow_1` with query: `q:Jordan, time:2025-05-12 14:00`.