COCORELI: Enforcing Execution Preconditions for Reliable Collaborative Instruction Following *

Swarnadeep Bhar[†]
IRIT
sbhar8597@gmail.com

Omar Naim[†]
Université de Toulouse
omar.naim.docs@gmail.com

Eleni Metheniti
ANITI
lenakmeth@gmail.com

Bastien Navarri
ANITI
navarri.bastien@gmail.com

Loïc Cabannes
ENS Paris-Saclay
loic.cabannes@gmail.com

Morteza Ezzabady
IRIT
morteza.ezzabady@irit.fr

Nicholas Asher
CNRS, IRIT
nicholas.asher@irit.fr

Abstract

Autonomous agents executing human instructions must operate reliably even when instructions are incomplete. While recent approaches improve detection of missing information, detection alone is insufficient: agents often proceed to execution even after recognizing underspecification, leading to incorrect or unsafe actions. We identify this failure as arising from a lack of coupling between detection and execution, and propose that reliable behavior requires enforcing missing information as a precondition for action. We instantiate this principle in COCORELI, a modular architecture that represents task structure, tracks missing information, and blocks execution until required details are resolved through targeted clarification. In COCORELI, detection and prevention are structurally coupled: detecting a missing parameter simultaneously blocks execution. We evaluate COCORELI in a controlled construction environment isolating underspecification and sequential execution. COCORELI makes execution reliable by imposing explicit task structure: instructions are represented as parameterized executable objects and execution under unresolved specifications is blocked by construction, eliminating hallucinated actions. In contrast, chain-of-thought, prompt-chaining, and ReAct-style reasoning may still execute under incomplete specifications despite high detection rates. The same representation supports abstraction and reuse, and generalizes to API workflow tasks on ToolBench. These results show that reliable collaborative execution under underspecified instructions requires architectural enforcement, not just model capability.

1 Introduction

Autonomous agents increasingly execute actions based on human instructions in collaborative settings such as robotics, manufacturing assistance, and tool-use systems. In these environments, instructions are often incomplete: humans rely on shared context and incremental communication, leaving essential task parameters unspecified. Existing systems frequently proceed by inferring missing details rather than acquiring them through interaction, producing plausible but incorrect actions. In collaborative construction tasks, this can lead to placing components in unintended locations or using incorrect parts. Such failures arise not from linguistic misunderstanding alone, but from acting without sufficient information to execute the task reliably.

Rather than attempting to model the full complexity of human dialogue, we focus on a minimal but necessary component of collaboration: resolving underspecified instructions that lack the information required for correct execution. Underspecification provides a controlled entry point into collaborative interaction because it can be systematically manipulated and directly determines whether an action can be carried out successfully. By isolating this factor, we study how agents detect missing information, request targeted clarifications, and incorporate the acquired information into subsequent actions, without addressing broader conversational phenomena such as negotiation or social reasoning.

Traditional single-shot prompting approaches, including chain-of-thought reasoning, rely on implicit inference to fill missing parameters, often producing confident but incorrect outputs. Recent agentic approaches attempt to mitigate these issues through prompt decomposition, tool use, and interaction loops, yet typically assume that instructions are executable as given or that missing details can be inferred from background knowledge. Without explicit mechanisms to represent task incompleteness and defer execution until required information is obtained, these systems may still act under insufficient specification, reflecting a decoupling between detecting missing information and prevent-

*Code available at: https://github.com/swarnadeep8597/Cocoreli

[†]These authors contributed equally to this work.

Proceedings of the 27th Annual Meeting of the Special Interest Group on Discourse and Dialogue, pages 516–535
August 2–5, 2026.

ing execution. Increasing model scale does not fundamentally address this limitation, as larger models tend to generate more plausible guesses rather than reliably detecting and resolving uncertainty.

In this paper, we investigate whether reliability under underspecification depends not only on model capability but also on how task structure and uncertainty are represented during execution.

In many collaborative scenarios, agents must operate in domains where task-specific training data is scarce or where new objects and procedures are introduced dynamically. Retraining or fine-tuning dialogue models for each new task is often impractical. Consequently, many approaches emphasize prompting-based methods and modular architectures that enable agents to interpret instructions and execute actions at runtime without additional training. This motivates designs that explicitly structure interaction and execution rather than depending solely on learned behavior.

To this end, we propose COCORELI, a modular architecture for collaborative task execution under partial, evolving, and incrementally specified instructions. COCORELI represents instructions as structured executable objects whose parameters may be known or missing. Missing elements trigger targeted clarification requests, and execution is deferred until sufficient information is available. The system maintains task-relevant state in external memory and supports reuse of previously constructed structures via abstraction, enabling reconstruction of earlier configurations in new contexts.

We evaluate COCORELI in a controlled collaborative construction environment designed to isolate key conditions that commonly arise in real collaborative tasks: incomplete task specification, evolving state with limited history, and reconstruction of previously observed structures. Controlled tasks allow systematic comparison of different architectural paradigms under well-defined uncertainty, rather than attributing performance differences to confounding factors present in natural datasets. To assess generality beyond spatial construction, we additionally evaluate the abstraction mechanism on ToolBench API tasks, which involve interpreting instructions into structured parameterized actions in a different domain.

Hypothesis. Reliable execution under underspecification requires not just detection of missing information, but architectural enforcement that prevents action until missing information is resolved. We test this by asking whether explicit structural coupling of detection and prevention improves reliability over approaches that rely on implicit inference, regardless of model scale. We do not claim COCORELI as a definitive solution; rather, we use it as a minimal instantiation of this property to show that prompting-based paradigms systematically lack it.

We summarize our contributions as follows:

1. **A structured mechanism for execution under underspecification.** We introduce COCORELI, an architecture that represents instructions as executable structures with explicit tracking of missing parameters, enabling targeted clarification and deferral of action until sufficient information is available.
2. **A controlled evaluation of collaborative execution.** We design a collaborative construction environment that isolates underspecification, sequential state changes, and structural reuse, allowing systematic comparison of different execution paradigms under well-defined uncertainty.
3. **Empirical analysis of execution reliability.** We compare structured parsing, chain-of-thought prompting, and prompt-chaining pipelines, showing that approaches lacking explicit mechanisms for managing incomplete specifications frequently produce incorrect actions.
4. **Evidence for structural reuse via abstraction.** We demonstrate that the proposed approach supports reconstruction of previously built structures in new contexts and transfers to ToolBench API tasks, suggesting applicability beyond spatial construction.

2 Related Work

Collaborative instruction following under partial and evolving specifications requires agents to ground language in an external environment, maintain state across interactions, and resolve missing information before executing actions. Prior work has addressed subsets of these challenges through dialogue management, clarification, tool use, and agentic architectures. We focus on approaches most relevant to execution under underspecification and their assumptions about when actions may occur.

Task-oriented dialogue systems. Classical task-oriented dialogue systems address incomplete instructions through slot filling and clarification ques-

System	Mem.	Grnding	G.Rails	Exec	CQ	Interact	P&P
ReAct	✗	✓	●	✓	✗	✗	✗
ART	✗	✓	●	✓	✗	✗	✓
SayCan	✗	✓	✓	✓	✗	✗	●
MapAgent	✗	✓	✓	✓	✗	✗	✓
Voyager	✓	✓	✓	✓	✗	✗	✓
ToolFormer	✗	✓	✗	✓	✗	✗	✓
ToolAlpaca	✗	✓	✗	✓	✗	✗	✓
PAL	✗	✓	✓	✓	✗	✗	✓
ClarifyCoder	✗	✗	✗	✗	✓	✓	●
ASK-TO-ACT	✗	✓	✓	✓	✓	✓	●
LEGOMem	✓	✓	●	✓	✗	✗	✓
CoALA (framework)	✓	✓	●	✓	●	✓	✓
COCORELI	✓	✓	✓	✓	✓	✓	✓

Table 1: Positioning COCORELI and prior *task-specific agent systems* within the CoALA perspective. All systems, including COCORELI, are instantiated in particular environments or domains. COCORELI differs in that it simultaneously operationalizes all dimensions in a single environment designed to stress-test grounding, clarification, abstraction, and execution. ✓: supported, ✗: not supported, ●: partially or implicitly supported.

tions within predefined domains. Early collaborative dialogue systems emphasized explicit representations of task state and shared information. TRAINS (ALLEN et al., 1995) modeled dialogue as collaborative plan construction, while the Information State framework (Larsson and Traum, 2000) formalized dialogue management as updates to structured representations of beliefs, goals, and dialogue context. Subsequent work on belief-state tracking and dialogue state tracking (DST) (Young et al., 2013; Jacqmin et al., 2022) established explicit maintenance of task-relevant information and uncertainty as a central component of task-oriented dialogue. More recent schema-guided approaches (Rastogi et al., 2019) represent tasks through explicit intents and slots, enabling generalization across services while preserving structured state tracking. Related work on frame-based common-ground representations (Blache and Houlès, 2021) similarly models dialogue through partially instantiated structures and uses clarification questions to resolve missing information.

Clarification in interactive systems. Recent work explores clarification question generation to improve task performance or recover from errors, including ASK-TO-ACT (Zhang et al., 2025) and ClarifyCoder (Wu et al., 2025). In these systems, clarification typically functions as a refinement mechanism within a reasoning pipeline, helping models improve task understanding before continuing execution. While effective for reducing ambiguity, clarification remains primarily a reasoning aid rather than an explicit control mechanism governing whether execution may proceed.

Agentic frameworks and reasoning agents. Recent agentic frameworks decompose tasks into sequences of reasoning and action. COALA (Sumers et al., 2024) characterizes agents in terms of memory, actions, and decision procedures, while agentic workflow approaches distribute tasks across specialized sub-agents (Sudarshan et al., 2024). Related LLM-based systems such as ReAct (Yao et al., 2023b), Reflexion (Shinn et al., 2023), ART (Paranjape et al., 2023), reinforcement-learning-based approaches (Ahn et al., 2022), debate-style reasoning (Lin et al., 2023), and memory-augmented planning (Kong et al., 2025) improve decision making through iterative reasoning, tool use, or self-correction. These approaches generally assume that instructions are sufficiently specified for execution or rely on post hoc recovery when errors occur.

Interactive and memory-augmented agents. Several systems incorporate interaction or memory to improve grounded instruction following. HELPER (Sarch et al., 2023) retrieves prior language–program pairs to support future tasks, while other approaches incorporate corrective feedback after execution (Mehta et al., 2024). Such systems improve adaptation and task performance through interaction and experience, but clarification or correction typically occurs after an action has already been proposed or executed.

Collaborative construction and abstraction. Collaborative construction and embodied task-learning environments highlight the challenges of translating language into executable action sequences (Kanitscheider et al., 2021; Lin et al., 2021). Voyager (Wang et al., 2024) demonstrates that reusable

skills can emerge through iterative interaction and memory, but stores learned procedures in relatively task-specific forms. More broadly, these systems emphasize planning, execution, and reuse in grounded environments without explicitly addressing underspecification as a first-class architectural concern.

Architectural capability comparison. Table 1 summarizes differences among these approaches across dimensions relevant to collaborative execution under underspecification, including persistent memory, environmental grounding, proactive clarification, execution guardrails, and modular composition. Across these lines of work, prior systems provide mechanisms for state tracking, clarification, memory, planning, or tool use, but typically treat missing-information detection and action execution as separate processes. COCORELI explicitly couples them: instructions are represented as executable task structures with unresolved parameters, missing information triggers targeted clarification, and execution is blocked until required parameters have been acquired. Our focus is therefore not only on improving clarification or reasoning, but on enforcing execution preconditions for reliable collaborative instruction following.

3 The COCOBOTS Task

We build upon COCOBOTS (Paetzel-Prüsmann et al., 2022), a collaborative construction environment designed to evaluate language agents on instruction following under underspecification, structured spatial reasoning, and interactive clarification. Patterned after the Minecraft Collaborative Building Task (Narayan-Chen et al., 2019), COCOBOTS departs from Minecraft-style environments through novel object types, altered interaction rules, and stricter physical constraints. Tasks are programmatically generated from a predefined inventory of parts and placement constraints, enabling controlled manipulation of instruction completeness and spatial complexity.

Task structure. As in the Minecraft task, COCOBOTS involves two agents with asymmetric information: an *Architect*, who has access to a target structure, and a *Builder*, who constructs the structure by following the Architect's natural language instructions. The Builder does not have access to the target structure except through the instructions issued during interactive conversation and the evolving scene state. Instructions may omit details, requiring the Builder to detect and resolve underspecification through interaction.

Cocobots and objects. The task is defined over a discrete $16 \times 16 \times 16$ three-dimensional grid, increasing spatial complexity relative to prior collaborative construction settings. Unlike Minecraft's uniform cubic blocks, COCOBOTS features nine synthetic part types with heterogeneous spatial properties, including single-cell components (e.g., screws, nuts, washers) and multi-cell components (e.g., bridges spanning two adjacent cells). Valid placement requires reasoning about orientation, spatial extent, and partial coordinate specifications.

Physical and execution constraints. COCOBOTS enforces strict physical constraints inspired by real-world assembly, including gravity and support dependencies. Incorrect placements are not permitted and cannot be corrected post hoc. As a result, agents must resolve ambiguities and ensure that instructions are sufficiently specified before executing actions.

Instruction characteristics. Architect instructions vary in specificity and form, ranging from fully specified commands to underspecified or relational descriptions (e.g., relative position, orientation, or part attributes). Instructions may also introduce previously unseen composite structures, requiring the Builder to interpret descriptions relative to the evolving scene state and to request clarification when required information is missing.

Design rationale. COCOBOTS addresses key limitations of existing collaborative benchmarks: (i) assumptions of complete specifications or permissive error correction, (ii) simplified spatial reasoning with uniform components, and (iii) limited evaluation of abstraction and reuse. Tasks are defined independently of any particular architecture and require correct execution given only the available information.

By combining underspecified language, structured spatial constraints, and irreversible execution errors, COCOBOTS provides a controlled testbed for studying collaborative instruction following.

Figure 2 contrasts COCOBOTS with Minecraft-based tasks. While retaining an intuitive spatial setting, it introduces sufficient novelty and constraint to require genuine interaction, abstraction, and clarification.

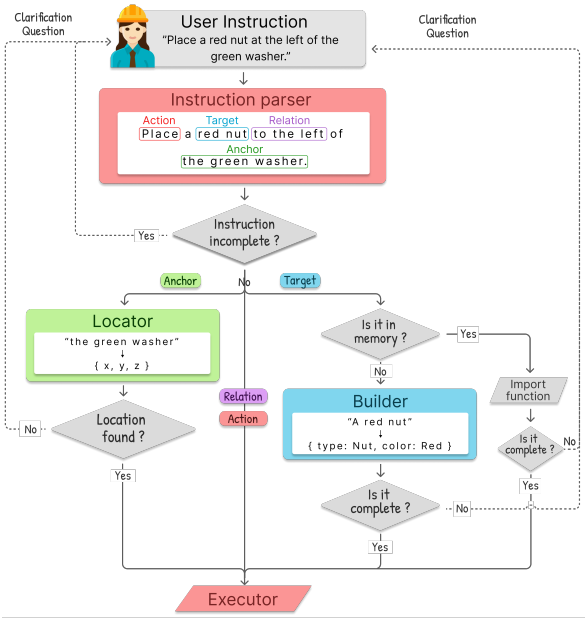


Figure 1: COCORELI, Our Structured Agentic Architecture

4 COCORELI: A Structured Agentic Architecture

COCORELI is a modular agentic architecture designed for collaborative instruction following under underspecified natural language commands. The architecture addresses three challenges highlighted by the COCOBOTS benchmark: (i) detecting missing or ambiguous information before execution, (ii) grounding relational language in structured spatial representations, and (iii) learning compositional abstractions that generalize across instances. Figure 1 provides an overview.

Architectural principles. Early experiments showed that monolithic prompting with a single LLM is brittle: long prompts reduce reliability and models often infer missing attributes instead of requesting clarification. COCORELI therefore decomposes the task into six specialized components: **Instruction Parser**, **Locator**, **Builder**, **External Memory**, **Discourse Module**, and **Executor**. Each component uses the same underlying LLM (LLaMA-3.1 8B; Meta, 2024) but operates with a specialized prompt and structured inputs.

A key design decision is to represent objects and actions as typed JSON structures whose fields are initialized to `null`. As dialogue proceeds, these fields are filled with information extracted from instructions or clarification responses. Execution is blocked whenever required fields remain unresolved; the system instead invokes the **Discourse Module** to generate a targeted clarification question. This mechanism prevents implicit guessing and enforces explicit resolution of missing information before execution.

Grounding and execution. Grounding proceeds through a sequence of specialized modules. The **Instruction Parser** first identifies the object types referenced in the instruction and extracts preliminary attributes from the language. This step filters the relevant object schemas, avoiding prompt bloat from passing all possible part representations to subsequent components.

The filtered object representations are then processed by two complementary modules. The **Builder** resolves object-specific attributes such as orientation or part configuration, while the **Locator** interprets spatial descriptions and maps relational expressions to candidate coordinates in the environment. Because different objects support different placement strategies (e.g., bridges span ranges of cells while washers occupy single coordinates), the Locator applies object-specific placement rules when interpreting spatial constraints. Missing object attributes are handled by the Builder and missing spatial parameters by the Locator; unresolved fields remain `null`, triggering clarification.

Abstraction and memory. COCORELI includes an optional **External Memory** that can store previously constructed structures as relational graphs whose nodes represent parts and whose edges encode spatial relations. When a structure is committed for reuse, an abstraction operation separates structural patterns from instance-specific parameters such as color or absolute position. These abstractions allow previously constructed configurations to be reproduced in new contexts by instantiating them with different parameters.

Execution. Once object attributes and spatial parameters are fully specified, the **Executor** converts the completed representation into a concrete JSON action and validates it against environmental constraints before placement. Errors are therefore avoided through clarification and structured grounding rather than corrected after the fact.

Scope. Although evaluated in a collaborative construction environment, the central mechanisms of COCORELI where explicit detection of underspecification, clarification-driven interaction, structured grounding, and abstraction are not specific to physical manipulation. COCORELI enforces a coupling between detection and execution: incomplete task

representations cannot be executed. This contrasts with prompting-based approaches, where detection and execution remain decoupled and actions may still be produced under uncertainty. These mechanisms capture a minimal requirement for reliable execution under underspecified instructions: missing task information should be resolved before acting.

5 Experiments for COCOBOTS tasks

5.1 Tasks and baselines

We evaluate COCORELI and the baselines on a set of COCOBOTS tasks designed as *diagnostic probes* of capabilities required for collaborative instruction following. The goal is not to construct a large-scale benchmark but to isolate common failure modes of language agents, particularly the tendency to execute actions despite incomplete instructions and to examine whether structured parsing and explicit task representations mitigate these failures.

The tasks are arranged in increasing complexity and progressively introduce additional requirements such as sequential state tracking, underspecification handling, and abstraction.

(i) **Single fully-specified instructions.** A single instruction specifying placement of one part with complete information.

(ii) **Instruction sequences.** Multiple fully specified instructions describing a sequence of placements.

(iii) **Complex structure construction.** Instruction sequences describing multi-part shapes that require maintaining spatial relations across steps.

(iv) **Underspecified instructions.** Instructions in which one or more required parameters are omitted, requiring the system to detect missing information and avoid execution until it is resolved.

(v) **In-context abstraction and reuse.** After constructing a complex structure from instructions, the system must store the resulting configuration and reproduce it in a new context (e.g., at a different location or with different attributes). This task probes whether systems can form reusable structural representations from interaction history or whether explicit memory mechanisms are required.

Tasks (i)–(ii) evaluate instruction parsing and sequential execution. Task (iii) introduces compositional spatial reasoning over multi-part structures. Task (iv) directly tests the ability to detect and resolve underspecified instructions through clarification. Task (v) evaluates whether systems can abstract previously constructed structures and reuse them in new contexts.

Taken together, these tasks allow us to examine how different architectural choices affect execution reliability as task complexity increases. They also function as *task-based ablations*, revealing which capabilities are necessary for each level of difficulty. Additional task details are provided in Appendix F.

Baselines We compare three architectural paradigms for instruction execution that differ in how strongly task structure is enforced.

Single-LLM prompting (CoT). The first baseline follows the monolithic prompting paradigm in which a single language model is responsible for interpreting instructions, reasoning about missing information, and generating the final action. We use chain-of-thought (CoT) prompting to encourage intermediate reasoning before producing the final placement specification. This setup reflects the common end-to-end approach where the entire task is solved within a single prompt without explicit intermediate control.

We evaluate this baseline using Claude-3.5 Sonnet (Anthropic, 2024) and GPT-4.1 (OpenAI, 2025). For tasks requiring reuse or abstraction, previously issued instructions must be repeated in the prompt since the model has no external memory. Preliminary experiments with LLaMA-3.1-8B showed substantially lower accuracy, so we report results for stronger models in this paradigm.

Agentic LLM decomposition. The second baseline represents an agentic design in which the task is decomposed into subtasks handled by specialized prompts, while each subtask is still solved end-to-end by an LLM. In this setup, the model executes instructions step-by-step for Tasks (i)–(iv), and for abstraction (Task v) is prompted to translate sequences of placement instructions into a reusable Python function that captures the structure of the instruction sequence. If the generated code is correct, reconstructing the structure reduces to invoking this function with new parameters.

This baseline therefore introduces modular task decomposition while retaining LLM-driven execution, without explicitly coupling clarification generation with execution control. We implement this approach using LLaMA-3-70B (Meta, 2024).

Structured parsing architecture. COCORELI

represents a third design paradigm in which subtasks are further decomposed into LLM-based interpretation and deterministic execution components. Instead of solving subtasks end-to-end, the LLM primarily produces structured representations that are validated and executed by specialized modules. This allows explicit detection of missing parameters and controlled clarification before execution.

Comparison scope. The goal of these baselines is not exhaustive optimization but to represent common architectural paradigms and test whether improved reasoning and detection prevent execution under incomplete specifications. The single-LLM baseline reflects monolithic prompting, the agentic baseline reflects modular decomposition with LLM-driven components, and COCORELI represents structured parsing with explicit task representations. Our comparison therefore examines how increasing levels of structural control affect execution reliability under underspecified instructions.

Further implementation details for the baselines and COCORELI are provided in Appendix A.

6 Results on COCOBOTS tasks

Results of Tasks (i) and (ii). Fully specified single-part instructions were handled reliably by all systems, with only minor errors in coordinate recovery for the CoT baselines. Sequences of two instructions (Task ii) proved more challenging, particularly for the single-LLM baselines, which showed reduced accuracy in identifying the properties of the second object. As discussed further in Task (iv), this difficulty becomes more pronounced when the second instruction is underspecified. Detailed results for Tasks (i) and (ii) are reported in Tables 6 and 7 in Appendix B.

Constructing complex shapes (Task iii). Task (iii) evaluates the ability to follow instruction sequences describing multi-part structures. A strict metric considers a shape correct only if all instructions are executed correctly. However, because complex structures consist of multiple instructions, a more informative measure is the proportion of correctly executed steps within each structure. Table 2 in Appendix J.1 reports this step-level accuracy.

Under this metric, COCORELI achieves the highest overall accuracy (78.57%), outperforming both the CoT baselines and the agentic baseline. Notably, COCORELI is the only system that partially reconstructs the most complex structure (the "Moroccan bridge"), which requires both abstraction and subsequent reference to the structure (see Task (v)). None of the other systems were able to execute this correctly. These results suggest that explicit task decomposition and structured execution improve robustness when instructions involve multiple spatial relations.

System	Step Acc (%)	Shapes Fully Built (%)
GPT-4.1 (CoT)	54.67	40
Claude 3.5 (CoT)	69.04	60
Agentic LLM	69.04	70
COCORELI	**78.57**	50

Table 2: Performance on complex shape construction (Task iii). Step accuracy measures the proportion of correctly executed instructions across shapes. The full per-shape breakdown is provided in Appendix J.1.

Underspecified instructions (Task iv). Task (iv) evaluates the ability to detect and resolve missing information; all metrics are computed over underspecified inputs. We first test underspecified single-part instructions, where systems must detect missing parameters by returning *null* values and, when possible, generate a clarification question (CQ).

As shown in Table 3, detection performance varies across models, with single-LLM baselines often failing to identify missing parameters. Only the agentic baseline and COCORELI can issue clarification questions. Once the missing information is provided, both systems correctly incorporate it without hallucinations.

We next evaluate underspecified instructions involving two parts (Table 4). COCORELI reliably detects missing information, asks clarification questions, and updates the parse without hallucinations. In contrast, CoT-style prompting infers missing attributes and hallucinates part properties. ReAct (implemented over GPT-4.1) and the agentic baseline improve detection and generate clarification-like queries, but do not enforce execution constraints, often proceeding under incomplete specifications and leading to persistent hallucinations. This shows that detection alone is insufficient, as baseline systems often infer missing attributes, leading to execution on incorrect but fully specified parses. Additional analysis of user burden is provided in Appendix I.

In-context abstraction and reuse (Task v). Task (v) evaluates whether systems can abstract a structure from instructions and reproduce it in later interactions. Models first construct nine complex structures and are then asked to recreate them without restating the original instructions. A reproduced

Shape	GPT	Claude	Agentic	COCORELI
Detected	27.2	49.4	**100**	**100**
CQs Asked	N/A	N/A	**100**	**100**
Correct (CQ)	N/A	N/A	**100**	**100**
Hallucinations:				
On Parts	20.0	0.0	0.00	0.00
On Colors	2.1	2.1	34	0.00
On Location	100	38.9	51.9	0.00

Table 3: Accuracy (%) on Task (iv): underspecified instructions describing one part.

Shape	GPT	Claude	Agentic	ReAct	COCORELI
Detected	54.5	53.5	98.5	96.5	96.5
CQs Asked	N/A	N/A	98.5	96.5	**100**
Correct (CQ)	N/A	N/A	**100**	96.5	**100**
Hallucinations:					
Parts	100	100	97.5	0.5	0.00
Colors	45.5	46.5	44.6	3.0	0.00
Location	100	100	78.2	1.5	0.00

Table 4: Accuracy (%) on Task (iv): underspecified instructions describing two parts. Part hallucination is reported for completeness; part identity is often inferable from context.

structure R is considered correct if there exists a bijection f between the parts of R and those of the original structure O that preserves all relative spatial relations.

Table 5 shows that all systems reproduce simple structures successfully. The agentic baseline fails on medium-sized shapes (16–18 parts), while the CoT baselines succeed on these cases but fail on the most complex structures. COCORELI reconstructs all evaluated structures, including the most complex ones.

The differences in performance reflect the nature of the tasks each system must solve. In the CoT and agentic baselines, abstraction requires the model to infer a reusable procedure from a sequence of placement instructions, which involves planning and program synthesis. In particular, the agentic baseline must generate Python code that captures the structure of the instruction sequence, and reconstruction succeeds only if this generated program is correct. In contrast, COCORELI treats abstraction as a structured parsing problem. The system extracts placement information from instructions and represents the resulting configuration as a relational graph with parts as nodes and spatial relations as edges. Reconstructing a structure reduces to re-instantiating this graph with new parameters. The higher accuracy observed for COCORELI thus reflects explicit structural representations rather than superior reasoning ability. See Appendix B.

7 Learning new functions in context in ToolBench

Shape Category	GPT	Claude	Agentic	COCORELI
Simple (4 shapes)	4/4	4/4	4/4	4/4
Medium (3 shapes)	3/3	3/3	1/3	3/3
Complex (2 shapes)	0/2	0/2	0/2	2/2

Table 5: Summary results for Task (v): abstraction and reproduction of novel shapes. Full per-shape results are provided in Appendix J.3.

To examine whether the abstraction mechanism of COCORELI transfers beyond spatial construction, we evaluate it on the ToolBench dataset (Qin et al., 2024). ToolBench contains workflows composed of API calls that access external services. While this domain differs from COCOBOTS, both settings require mapping natural language instructions to structured, parameterized actions that can later be reused with new arguments. We use ToolBench because it provides workflow demonstrations that can be abstracted and re-instantiated with new parameters, allowing us to directly evaluate structural reuse beyond the spatial domain.

ToolBench does not contain underspecified instructions and therefore does not exercise the clarification mechanism. Instead, the experiment isolates the abstraction capability: given a workflow and new input parameters, the system must reproduce the correct function calls with updated arguments. Because the task concerns structured reuse rather than interactive execution, we compare COCORELI only with single-LLM CoT baselines.

Methodology After preprocessing the workflows (Appendix L), we provide the system with (i) the original user instruction and corresponding API call sequence and (ii) new parameter values. COCORELI parses the workflow instruction with the Instruction Parser and produces a structured representation `workflow_i`. When given the cue *this is a workflow_i*, the Builder abstracts this representation and stores the template in External Memory. The stored template can then be instantiated with new arguments to produce the updated API calls. In contrast, the CoT baseline (Claude 3.5 Sonnet) must infer the workflow pattern directly from the prompt and regenerate the calls with modified parameters.

Results On 100 ToolBench workflows, the CoT baseline (Claude) achieved 36.4 precision, 26.9 recall, and 30.9 F1, whereas COCORELI achieved 100 on all three metrics. Precision measures the

proportion of generated API calls that match the expected workflow, while recall measures the proportion of expected calls correctly reproduced.

These results should not be interpreted as demonstrating superior reasoning. In ToolBench the baseline must infer workflow structure through planning and pattern induction, whereas COCORELI converts the task into structured parsing followed by deterministic abstraction via its relational representation. The experiment therefore illustrates how explicit structural representations and external memory enable reliable workflow reuse rather than indicating that COCORELI performs a harder task.

Overall, reliable agent execution benefits from explicit mechanisms for both clarification of missing task parameters and reusable task abstractions.

8 Conclusion

Reliable execution under underspecified instructions requires more than detecting missing information. Agents must also prevent execution until the information necessary for task completion has been acquired. To investigate this principle, we introduced COCORELI, a modular architecture that represents instructions as executable task structures, explicitly tracks unresolved parameters, and couples clarification with execution control.

Experiments in COCOBOTS show that enforcing execution preconditions through structured representations and targeted clarification prevents hallucinated actions under incomplete specifications, whereas prompting-based and agentic baselines may still execute despite detecting missing information. We further demonstrated that the same representations support abstraction and reuse beyond spatial construction through transfer to ToolBench workflows.

Taken together, our results suggest that reliable instruction following under incomplete task specifications depends not only on model capability, but also on architectural mechanisms that explicitly represent task requirements and enforce their satisfaction before execution. We hope that COCORELI and COCOBOTS provide useful foundations for future work on collaborative agents operating under incomplete information.

Acknowledgments

This work was supported by the FRAE-RAKEL project (R093–FRAE–RAKEL), the ANR project COCOBOTS (ANR-21-FAI2-0005), and the AI Cluster ANITI (ANR-19-PI3A-0004). The experiments reported in this paper were performed using HPC resources provided by CALMIP (Grants 2016-P23060 and M24047).

Limitations

COCORELI focuses on reliability under underspecified instructions rather than providing a complete architecture for collaborative agents. Several limitations remain.

First, our approach assumes that tasks can be represented through structured schemas. In our experiments, API functions and object specifications are expressed in JSON, which enables explicit detection of missing parameters. Real-world systems may require additional preprocessing or schema design to convert raw tool interfaces or environment states into such structured representations. Moreover, identifying which recurring patterns in demonstrations or interaction histories should be abstracted into reusable task structures remains a challenging problem in its own right. Our work assumes that such reusable patterns can be identified and focuses instead on their structured representation and subsequent reuse. Second, our guarantees apply only when underspecification can be represented within a known task schema. COCORELI does not address cases where instructions are fully specified but incorrectly parsed, where users refer to entities or actions that fall outside the schema, or where ambiguity arises from pragmatic intent rather than missing task parameters. These forms of uncertainty require richer semantic representations, schema adaptation mechanisms, and more sophisticated models of user intent. Although COCOBOTS is intentionally controlled to isolate underspecification, sequential state tracking, and structural reuse, future work should evaluate these principles in richer collaborative environments involving more complex planning and grounding requirements.

Third, our conversational component models only a narrow form of dialogue: clarification for resolving missing task parameters. It does not address other collaborative discourse phenomena such as corrections, negotiation of goals, or extended multi-turn reasoning. Handling richer interaction patterns would require more sophisticated dialogue management mechanisms.

An additional limitation concerns the experimental comparison. Our baselines represent three commonly used architectural paradigms rather than matched implementations on a shared backbone.

Although our goal is to evaluate the effect of architectural design rather than optimize individual prompting strategies, future work should compare COCORELI against equally engineered CoT or agentic pipelines using the same underlying language model. Such experiments would further isolate the contribution of the execution architecture from differences in prompt engineering effort and model capabilities.

Finally, COCORELI does not include an explicit planning module for decomposing high-level goals into sequences of actions. In our experiments, instructions are assumed to be provided by an Architect agent, and integrating structured clarification and abstraction mechanisms with planning remains an open challenge.

Despite these limitations, our experiments highlight a necessary architectural capability for reliable agent execution under incomplete task specifications. Systems that lack explicit mechanisms for detecting missing task parameters and representing reusable task structures tend to rely on implicit inference, which frequently leads to incorrect or inconsistent actions. Our results therefore suggest that clarification and structured abstraction are important architectural components for reliable execution under incomplete task specifications.

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, and 26 others. 2022. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. *Preprint*, arXiv:2204.01691.

JAMES F. ALLEN, LENHART K. SCHUBERT, GEORGE FERGUSON, PETER HEEMAN, CHUNG HEE HWANG, TSUNEAKI KATO, MARC LIGHT, NATHANIEL MARTIN, BRADFORD MILLER, MASSIMO POESIO, and DAVID R. TRAUM. 1995. The trains project: a case study in building a conversational planning agent. *Journal of Experimental & Theoretical Artificial Intelligence*, 7(1):7–48.

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

Philippe Blache and Matthis Houlès. 2021. Common ground, frames and slots for comprehension in dialogue systems. pages 33–43.

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.

David Harel. 1984. *Dynamic Logic*, pages 497–604. Springer Netherlands, Dordrecht.

Léo Jacqmin, Lina M. Rojas Barahona, and Benoit Favre. 2022. "do you follow me?": A survey of recent approaches in dialogue state tracking. In *Proceedings of the 23rd Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 336–350, Edinburgh, UK. Association for Computational Linguistics.

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.

Staffan Larsson and David R. Traum. 2000. Information state and dialogue management in the trindi dialogue move engine toolkit. *Nat. Lang. Eng.*, 6(3–4):323–340.

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.

Stephanie Lin, Jacob Hilton, and Owain Evans. 2021. Truthfulqa: Measuring how models mimic human falsehoods. *arXiv preprint arXiv:2109.07958*.

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, and 3 others. 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.

Meta. 2024. The Llama 3 Herd of Models. *Preprint*, arXiv:2407.21783.

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.

Omar Naim, Swarnadeep Bhar, Jérôme Bolte, and Nicholas Asher. 2026a. Ssa: Improving performance with a better scoring function. *Preprint*, arXiv:2508.14685.

Omar Naim, Jérôme Bolte, and Nicholas Asher. 2025a. Analyzing limits for in-context learning. *arXiv preprint arXiv:2502.03503*.

Omar Naim, Guilhem Fouilhé, and Nicholas Asher. 2025b. Re-examining learning linear functions in context. In *German Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 104–117. Springer.

Omar Naim, Krish Sharma, Niyar R Barman, and Nicholas Asher. 2026b. Tell-tale: Task efficient llms with task aware layer elimination. *Preprint*, arXiv:2510.22767.

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].

Maike Paetzel-Prüsmann, Julie Hunter, Kranti Chalamalasetti, Kate Thompson, Alexandros Nicolaou, Ozan Güngör, David Schlangen, and Nicholas Asher. 2022. Conversational programming for collaborative robots. In *ICRA Workshop on Collaborative Robots and Work of the Future (ICRA 2022 CoR-WotF)*, pages 1–5.

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*.

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*.

Abhinav Rastogi, Xiaoxue Zang, Srinivas Kumar Sunkara, Raghav Gupta, and Pranav Khaitan. 2019. Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset. *arXiv preprint arXiv:1909.05855*.

Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Garðar Ingvarsson, Timon Willi, Akbir Khan, Christian Schroeder de Witt, Alexandra Souly, Saptarashmi Bandyopadhyay, Mikayel Samvelyan, Minqi Jiang, Robert Tjarko Lange, Shimon Whiteson, Bruno Lacerda, Nick Hawes, Tim Rocktäschel, Chris Lu, and Jakob Nicolaus Foerster. 2023. JaxMARL: Multi-agent RL environments in JAX. In *Second Agent Learning in Open-Endedness Workshop*.

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.

Theodore Sumers, Shunyu Yao, Karthik R Narasimhan, and Thomas L. Griffiths. 2024. Cognitive architectures for language agents. *Transactions on Machine Learning Research*. Survey Certification, Featured Certification.

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.

Jie JW Wu, Manav Chaudhary, Davit Abrahamyan, Arhaan Khaku, Anjiang Wei, and Fatemeh H. Fard. 2025. Can code language models learn clarification-seeking behaviors? *Preprint*, arXiv:2504.16331.

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.

Steve Young, Milica Gašić, Blaise Thomson, and Jason D. Williams. 2013. Pomdp-based statistical spoken dialog systems: A review. *Proceedings of the IEEE*, 101(5):1160–1179.

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.

Xuan Zhang, Yongliang Shen, Zhe Zheng, Linjuan Wu, Wenqi Zhang, Yuchen Yan, Qiuying Peng, Jun Wang, and Weiming Lu. 2025. Asktoact: Enhancing llms tool use via self-correcting clarification. *Preprint*, arXiv:2503.01940.

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 On the Nature of the Architectural Contribution

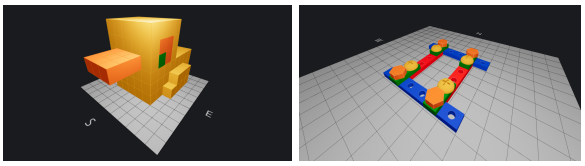
The empirical comparisons in this paper may be interpreted as performance comparisons across systems of different scales. We clarify that the primary contribution is architectural rather than performance-driven.

The central claim is not that COCORELI outperforms larger models, but that it enforces a structural coupling between detecting missing information and executing actions. In COCORELI, incomplete task representations cannot be executed: if required parameters remain unresolved, execution is blocked. As a result, the absence of hallucinated actions under underspecification follows from this execution model, rather than from improved detection or reasoning capability.

The baseline comparisons are intended to illustrate that this property is absent in prompting-based paradigms. In such systems, detection of missing information and action generation are decoupled processes: even when missing information is identified during reasoning, actions may still be produced based on inferred values. This behavior reflects architectural design choices rather than limitations of model capability.

The difference in model scale between COCORELI (LLaMA-3.1 8B) and the CoT baselines (GPT-4.1, Claude-3.5-Sonnet) should therefore be interpreted in this context. The structural enforcement of execution preconditions in COCORELI operates independently of model scale, while prompting-based approaches rely on model behavior to avoid incorrect execution. Such architectures could also benefit from complementary efficiency methods such as TALE, which improves task-specific performance while further reducing model size (Naim et al., 2026b).

We note that this guarantee applies to missing-parameter cases. Incorrect but fully specified representations remain possible and represent an important limitation of the current approach.



Minecraft Cocobots

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

C COCOBOTS-MINECRAFT

Figure 2 contrasts our environment with Minecraft-based tasks.

D Tasks 1 and 2

Evaluation Protocol. Evaluation is performed on structured representations (part, color, coordinates, relations) rather than raw text outputs. For multi-step outputs, only the final action corresponding to each instruction is evaluated, as intermediate actions may include state reconstruction. The second instruction is classified as independent if it specifies absolute coordinates, and dependent if it refers to a previously placed part (e.g., "on top", "left of"). For dependent placements, correctness is determined by relation type and anchor consistency rather than absolute coordinates. Coordinates follow a row–column convention with the top-left cell defined as (1,1).

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 COCOBOTS 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).

Results of tasks 1 and 2: 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 for the placement of an object. Table 6 shows the results. 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.

Our system performed well on Task (ii) as well 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.

For Task (ii), 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 7. The second part of the sequence may have either independent coordinates ("Independent") or dependent coordinates ("Dependent") to an anchor part ("Dep. Anchor").

E Task 3 and 5

Here are the full results on the more complex shapes, given in Table 8

Accuracy	GPT-4.1	Claude	Agentic	COCORELI
Part type	**100**	**100**	**100**	**100**
Part color	**100**	**100**	**100**	**100**
Coordinates	83.33	83.33	**100**	**100**
TOTAL	94.44	94.44	**100**	**100**

Table 6: Fine-grained accuracy results (%) on Task (i): Single-part placement.

Accuracy	GPT-4.1		Claude		Agentic		COCORELI	
	1st	2nd	1st	2nd	1st	2nd	1st	2nd
Part type	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**
Part color	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**
Coordinates	**100**	–	**100**	–	**100**	–	**100**	–
Independent	–	**100**	–	**100**	–	**100**	–	**100**
Dependent	–	**100**	–	**100**	–	**100**	–	**100**
Dep. Anchor	–	63.64	–	63.64	–	**100**	–	**100**
TOTAL	96.97		96.97		**100**		**100**	

Table 7: 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 (coordinates are not evaluated for dependent placements). While overall accuracy is high, performance on dependent anchor placement reveals persistent failures in executing relational instructions.

Here is the full results for the abstraction claims in Table 9

E.1 COCOBOTS task data

We provide detailed data for Tasks (i) and (ii) in Appendix D and focus here on the remaining task families.

Complex structure construction. Task (iii) evaluates the ability to follow sequences of instructions describing multi-part structures. Instructions may involve standard or complex parts and can specify either absolute or relational placements (e.g., "*place a horizontal row of four purple gaskets*"). The set of evaluated shapes is listed in Table 2; additional examples and visualizations are provided in Appendix J.1.

Underspecified instructions. Task (iv) evaluates handling of incomplete instructions. We construct two variants: underspecified single-sentence instructions (analogous to Task (i)) and underspecified two-sentence sequences (analogous to Task (ii)). Dataset construction details are given in Appendix J.2.

Learning abstract functions in context. Task (v) evaluates whether models can abstract a structure from instructions and reproduce it in later interactions. Systems first construct nine complex struc-

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 8: Accuracy (%) based on # correct steps /#total steps following instructions in Task (iii).

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

tures from instructions and are then asked to recreate them without explicit restatement of the original steps. The evaluated structures are listed in Table 5, with further details in Appendix J.3.

F Ablation Study

F.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 10, each removal leads to categorical collapse (0% 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 10: Controlled ablation results on representative subsets of the relevant task families. Each removal produces categorical collapse (0% 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 6, Table 7).
- **Complex multi-part instructions (Task iii).** Accuracy drops as implicit planning load increases, highlighting the limits of planning without a dedicated planner (Table 2).
- **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 3, Table 4).
- **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 5).

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.

G More Related Work

We define an agentic system as a triple $\mathcal{A} = (s_0, S, \rightarrow_A)$, with S a set of states, s_0 a special state representing the environment, and $\rightarrow_A$: $S \rightarrow S$ a non-empty typed set of labelled transitions S with A a set of action types. Agents are functions or relations (if agents are nondeterministic) between states, while states represent: (i) inputs from the environment to an agent, (ii) a repository of information about properties and objects, possible actions, as well as goals and constraints relevant to system tasks, (iii) representations pertinent to the task that one agent provides to another agent. We can compose agents together, as the set of transitions are closed under composition $a_1 \circ a_2$, non deterministic choice $a_1 \cup a_2$, iteration a^*, providing general programmable relations over functions.

Our agentic definition is thus a special case of a dynamic semantics used for programming languages (Harel, 1984), which enables sophisticated logical techniques for studying the logical properties of agentic systems. A

Our definition applies to an agentic system that takes a complex prompt of an LLM and divides it into a sequence of smaller prompts for different LLM agents. We can also easily represent memory as a state that an agent may operate on by adding new transitions or adding more facts, constraints and functions. An executor defines a transition on s_0. A decision procedure is a transition that outputs a state for the executor.

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., 2025a,b, 2026a; Mirzadeh et al., 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 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) extends 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 enhances 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. (2023) 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 COCOBOTS 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 COCOBOTS'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 COCOBOTS. Moreover, HELPER's feedback mechanism is reactive and post-execution, asking for corrections after acting, whereas COCOBOTS 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.

H Cost Analysis

Table 11 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 issue 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.

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 11: 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 (COCORELI 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 Table 2).

We do not report efficiency for the abstraction experiments in Table 5, 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.

I 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.

I.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 cognitive and operational load on the user.

J COCOBOTS dataset details

J.1 Task (iii)

For this task, every single instruction is fully specified. The instructions themselves are complex, drawing away from the simple template style approach we were following. 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 COCOBOTS parts. A detailed list and a visualization of the shapes in the COCOBOTS for this task is below. 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).

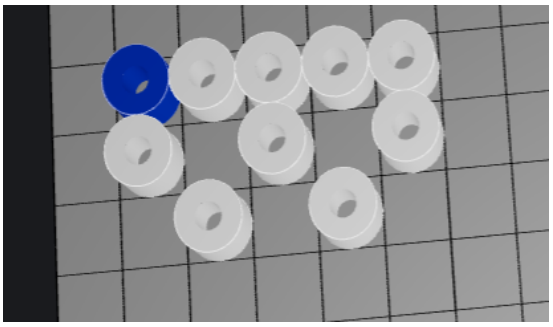


Figure 3: The B Shape

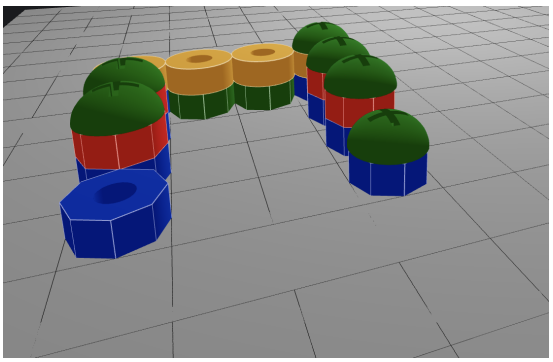


Figure 4: The Moroccan Bridge

J.2 Task (iv)

We detail here the two data sets of underspecified instructions. For the single sentence dataset, 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 two sentence dataset, 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.

J.3 Task (v)

We give details on the 9 structures for this task. 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).

A **sample** of human input to COCORELI 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

K 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
    $X \leftarrow$ 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
  $T[C] \leftarrow$ 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 component
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 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
$C_{Starts} \leftarrow$ 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
$S_x, S_y, S_z \leftarrow$ The bounding box for G
$S_G \leftarrow$ 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: $C_{val} \leftarrow$ From Chatter(new color)
Require: $P_{val} \leftarrow$ From Chatter and Locator(new Part)
Require: $S_{val} \leftarrow$ From Chatter(new Dimension)
  G' $\leftarrow$ Algo(3) with C_{val}, P_{val}
  Final $\leftarrow$ Algo(4) on G' with S_{val}
  end if
  if $C \leftarrow N$ and $P \leftarrow Y$ and $S \leftarrow Y$ **then**
Require: $P_{val} \leftarrow$ From Chatter and Locator(new Part)
Require: $S_{val} \leftarrow$ From Chatter(new Dimension)
  G' $\leftarrow$ Algo(3) with P_{val}
  Final $\leftarrow$ Algo(4) on G' with S_{val}
  end if
  if $C \leftarrow N$ and $P \leftarrow N$ and $S \leftarrow Y$ **then**
Require: $S_{val} \leftarrow$ From Chatter(new Dimension)
  G' $\leftarrow$ Algo(3) with P_{val}
  Final $\leftarrow$ Algo(4) on G' with S_{val}
  end if

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

L 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 instruction *"Schedule a meeting with Alex tomorrow at 3 pm and send a confirmation email."* is abstracted into the workflow `create_event(q,time)` → `send_email(q,message)`, which is stored in memory as `workflow_1`. At inference time, the stored workflow can be reused with new variables, e.g., `q: Jordan, time: 2025-05-12 14:00`, without relearning the workflow structure.

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. The nine structures used in Task (v) are: **A20 tower** (4 parts, 5 instructions), **C15 stack** (7 parts, 6 instructions;), **D21** (6 parts, 5 instructions), **X34** (5 parts, 5 instructions), **Square** (16 parts, 17 instructions), **Triad** (18 parts, 17 instructions), **Face** (47 parts), **I** (18 parts;), and **Skull** (62 parts, 63 instructions; Figure 5).

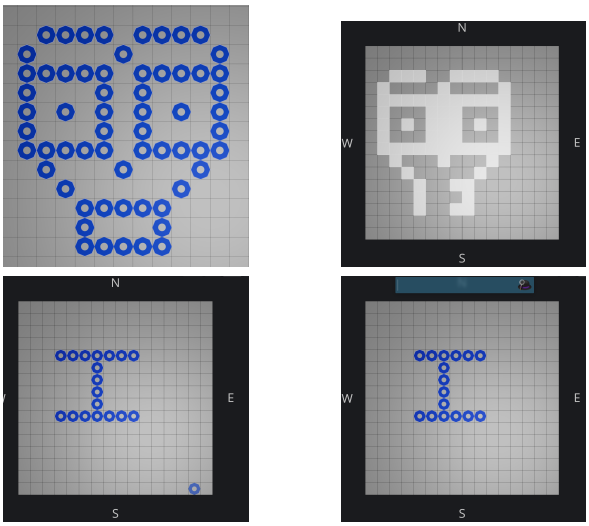


Figure 5: Representative failure cases on Task (v). Top: target Skull shape (left) and GPT-4.1 reconstruction (right). Bottom: target I shape (left) and GPT-4.1 reconstruction (right). Higher resolution versions are available in the repository.

M Failure cases of the models

We present two failure cases on Task (v) from the GPT-4.1 baseline in Figure 5. As discussed in Section 6 and shown in Table 5, the baseline was not able to reproduce the more complex shapes of the task from the dialogue history, but COCORELI was, through abstraction.