RoboScribe: Language-Guided Scripted Policy Generation for Robotic Manipulation

Yuer Tang Zhiyi Chen

Abstract

Collecting scripted trajectories is a necessary step in early-stage robotics research. Noiseless, controllable demonstrations enable clean experiments before moving to noisy human data. Yet writing scripted policies by hand requires researchers to spend hours learning simulator APIs, observation spaces, and low-level controller details. That is time better spent on actual research. We present RoboScribe, a language-guided pipeline that converts natural language task descriptions into executable low-level control policies for robotic manipulation in simulation. RoboScribe splits the problem into a human-reviewed phase design step and an autonomous generate-test-diagnose-revise code loop. We evaluate across four manipulation tasks of increasing difficulty and find that the system achieves 100% success on standard tasks (Lift, Stack) while exposing fundamental limits of LLM-generated code on tasks requiring precise geometric reasoning (NutAssembly, Door). These results suggest that the highest-leverage point for human input is at the strategy level, defining task phases, rather than at the code level. This motivates a pipeline where humans define what needs to happen and agents figure out how to do it.

1 Introduction

Scripted trajectory collection is a foundational step in robotics and embodied AI research. Before working with noisy human demonstrations, researchers rely on scripted policies to generate noiseless, distribution-controllable trajectories that are easier to debug, flexible across objects, and composable across tasks [7]. These properties make scripted data essential for validating algorithms, training initial policies, and studying trajectory distributions.

However, writing scripted policies is tedious and error-prone. A researcher must: (1) identify the correct observation keys for each environment; (2) implement a multi-phase state machine with precise positioning, gripper timing, and transition logic; (3) debug subtle failures like missed grasps or coordinate frame errors. Even experienced researchers spend hours per task—time that could be spent on their actual research questions.

Recent work has shown that large language models (LLMs) can generate code for robotic tasks [3], but existing approaches either operate at a high abstraction level using pre-built primitives [1, 3] or generate intermediate representations like reward functions that require separate optimizers [5, 6]. No existing tool lets a researcher describe a manipulation task in natural language and receive a working `get_action(obs)` policy for simulation. This gap leaves researchers manually coding the most tedious part of their workflow.

Scientific question. Beyond the practical tool, we investigate a deeper question: how does task geometric complexity affect LLM code generation success in low-level robotic control? Prior work suggests LLMs excel at high-level task decomposition but struggle with precise spatial reasoning [3, 4]. We test this directly by asking LLMs to generate complete low-level policies and examining where they succeed and fail across tasks of increasing geometric complexity. We observe that success rate correlates strongly with the geometric precision required: translational tasks (Lift, Stack) succeed reliably, while tasks involving quaternions, coordinate transforms, and multi-axis coordination (Door, NutAssembly) fail consistently. Furthermore, the diagnostic

system correctly identifies failures in every case, but the LLM cannot translate correct diagnoses into correct code fixes—suggesting the bottleneck is mathematical reasoning under physical constraints, not information availability. These observations motivate a hybrid architecture where human expertise enters at the strategy level (phase design) and domain constraints encode the geometric knowledge that LLMs cannot reliably derive.

2 Related Work

LLMs for robotic task planning. SayCan [1] and Inner Monologue [2] use LLMs as high-level planners that select from pre-trained low-level skills. Code as Policies [3] generates Python programs that compose perception APIs and control primitives. These systems restrict the LLM to task decomposition; low-level execution relies on pre-built skills or motion planners. RoboScribe operates at a lower level: the LLM generates the actual `get_action(obs)` control code.

LLM-generated reward and value functions. Language to Rewards [5] and Eureka [6] use LLMs to generate reward functions, which are then optimized via trajectory optimization or RL. VoxPoser [4] generates 3D value maps that guide a motion planner. In all cases, the LLM produces an intermediate representation—the policy comes from an optimizer, not from LLM code. RoboScribe generates policies directly, without a separate optimization step.

Automated demonstration generation. MimicGen [7] generates large-scale demonstrations by augmenting a small set of human demos. RoboGen [8] uses LLMs to generate tasks and supervision for RL training. Neither produces standalone scripted policies from language alone.

Gap. No existing system converts natural language task descriptions into executable scripted policies for robosuite-style simulation environments. RoboScribe fills this gap with a phase-design-first architecture that combines human strategy oversight with autonomous code generation and simulation-driven iteration.

3 Method

We build our agentic AI framework with interactive user interface, RoboScribe, which is a pipeline with two stages: Phase Design (human-in-the-loop) and Code Iteration (autonomous agent loop). Figure 1 shows the full architecture.

3.1 Stage 1: Phase Design

Given a natural language task description (e.g., "stack the red cube on the green cube"), RoboScribe proceeds through three steps:

Environment detection. The LLM selects the appropriate robosuite environment from a registry of supported tasks, matching the user's description to environment metadata.

Environment introspection. Before generating any code, RoboScribe spins up the selected environment in a subprocess and captures ground-truth information: all observation keys with their shapes, dtypes, and sample values; the source code of the `reward()`, `_check_success()`, and `staged_rewards()` functions via `inspect.getsource`. This introspection is injected into the LLM prompt, ensuring the model works from actual environment semantics rather than memorized or hallucinated APIs.

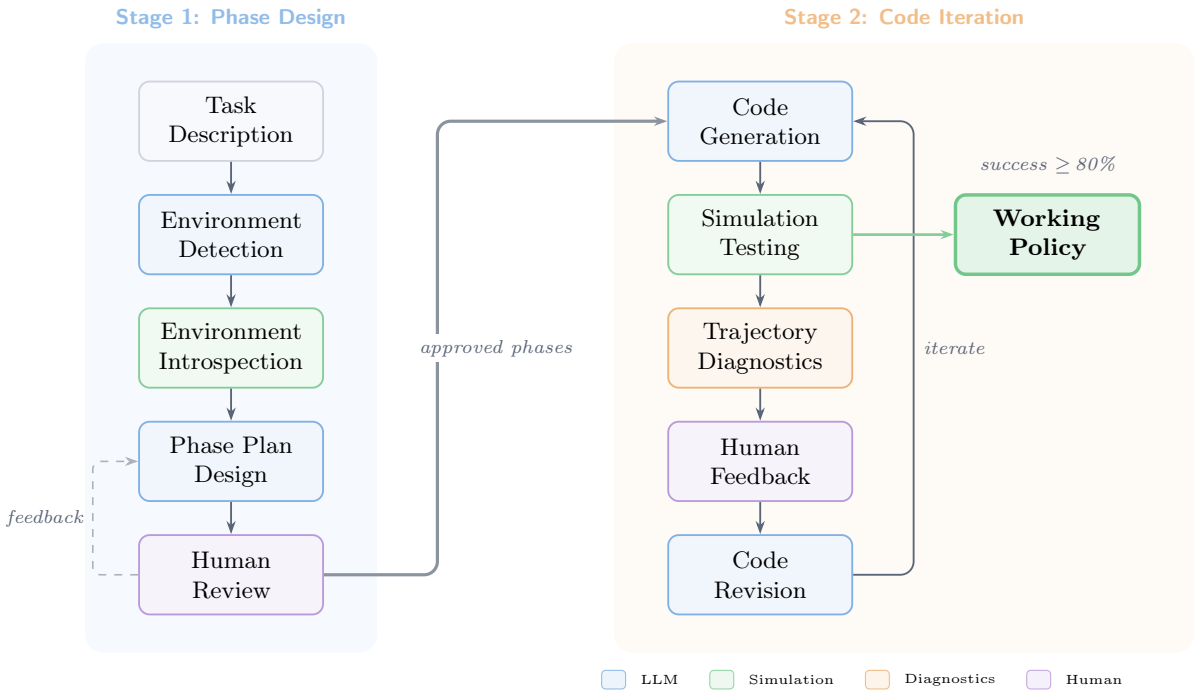


Figure 1: **RoboScribe pipeline.** Stage 1 (top left): the LLM auto-detects the environment, introspects observation/action spaces, and designs a phase plan for human review. Stage 2 (top right): a tool-use agent generates code from the approved plan, tests in simulation, diagnoses failures from trajectory metrics, and iterates with optional human feedback until the success threshold is met.

Phase plan generation. The LLM decomposes the task into a sequence of control phases, each specifying: a name, goal, control strategy, and exit condition. For example, a Stack task produces: APPROACH_A → LOWER_A → GRASP_A → LIFT_A → MOVE_TO_B → LOWER_TO_B → RELEASE → RETREAT. The user reviews this plan and may approve, edit, or regenerate with feedback. This is the primary human intervention point.

3.2 Stage 2: Code Iteration

Code generation. The LLM generates a complete Python policy with `get_action(obs)` and `reset()` functions, implementing the approved phases as a state machine. The system prompt includes domain constraints: OSC_POSE action format, proportional control patterns, gripper timing requirements, and observation access patterns.

Simulation testing. The generated policy runs in a robosuite subprocess for N episodes with randomized initial conditions. The runner captures per-episode success, rewards, and trajectory snapshots.

Trajectory diagnostics. When a test fails, the diagnostics module formats per-step trajectory metrics—distances to target objects, gripper state, object heights, environment-specific measurements—into a structured report. Figure 2 shows how information from multiple sources feeds into the diagnosis. The LLM reasons about what went wrong from these measurements.

Code revision. The agent receives the trajectory diagnostics and generates a revised policy. It has access to four tools throughout the loop. It uses `test_policy` to run the policy in simulation and get back the

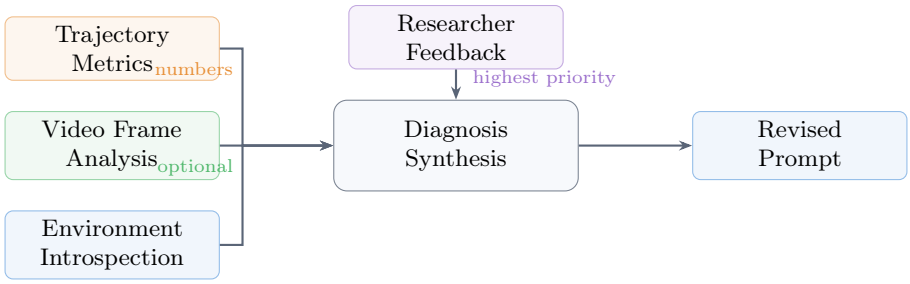


Figure 2: Diagnosis information flow. Multiple sources feed into a synthesized diagnosis. When sources conflict, human feedback takes priority, followed by trajectory measurements, then automated analysis.

success rate, trajectory metrics, and video frames. If something looks off, it calls `inspect_env` to re-check observation keys, shapes, and data types against the live environment. When it needs to understand reward logic or controller behavior, it reads the robosuite source directly with `read_robosuite_source`. Once satisfied, it calls `submit_policy` to declare the policy complete. The loop runs until success reaches 80% or the turn limit is hit.

3.3 Domain Constraints

A key design decision is the injection of domain-specific constraints into the LLM prompt. These encode knowledge that LLMs consistently fail to derive from first principles:

- **Action format**: OSC_POSE actions are 7-dim deltas $[dx, dy, dz, d\alpha_x, d\alpha_y, d\alpha_z, \text{gripper}]$, clipped to $[-1, 1]$.
- **Gripper timing**: The gripper needs 10–15 simulation steps to fully close; policies must wait before lifting.
- **Observation types**: Some observation values are scalars, not arrays—`obs['key'][0]` on a scalar crashes silently.
- **Control patterns**: Proportional control with gain 5–15 for position; PID controllers with correct reset semantics (reset once per phase transition, not every step).
- **Environment-specific**: Handler offset geometry for NutAssembly, hinge vs. handle angle semantics for Door.

These constraints are not the policy—they are guardrails that prevent classes of errors the LLM would otherwise make. The LLM still generates novel code for each task.

3.4 System Implementation

RoboScribe is implemented as a decoupled, real-time web application that separates backend orchestration from frontend interaction. The backend is written in Python 3 and served via a FastAPI application (deployed with Uvicorn), exposing REST endpoints for session management, configuration, and user feedback, alongside a persistent WebSocket channel for streaming intermediate results during long-running agent execution (e.g., simulation frames, tool logs, diagnostic outputs, and completion events).

The frontend is a single-page application built in React (JSX) and bundled with Vite. The interface is styled using lightweight inline and component-scoped CSS, avoiding reliance on server-side rendering frameworks such as Next.js. Communication between the browser and backend occurs over HTTP and WebSocket protocols, with CORS enabled for local development.

Core functionality, including policy generation, environment introspection, and robosuite simulation rollouts, is implemented within the same Python codebase. Numerical computation is handled via NumPy, while large language model interactions are supported through provider SDKs (e.g., OpenAI and Anthropic). Optional multimedia dependencies such as OpenCV and imageio enable video recording, frame streaming, and vision-assisted failure diagnosis.

4 Getting Started

RoboScribe installs with a single command: `pip install roboscribe[sim]`. It requires Python 3.10 and any LLM API key. From there, a researcher types a task description and the system handles the rest. Running `roboscribe run "stack the cubes"` kicks off the full pipeline from the command line. For a richer experience, `roboscribe serve` opens a web UI with live simulation video and a feedback box at every iteration. The output is a standalone `.py` policy file that works directly with `roboscribe test` and `roboscribe record`. No framework lock-in. The system supports four LLM backends: OpenAI, Anthropic, Qwen, and DeepSeek.

5 Evaluation

We evaluate RoboScribe along three dimensions: task complexity vs. LLM capability where we will specifically also focus on showing you the geometry complexity, human behavior shown by the learning rate.All runs use Qwen, the Panda robot with OSC_POSE controller, and no human feedback during code iteratio.

5.1 Task Complexity vs. LLM Capability

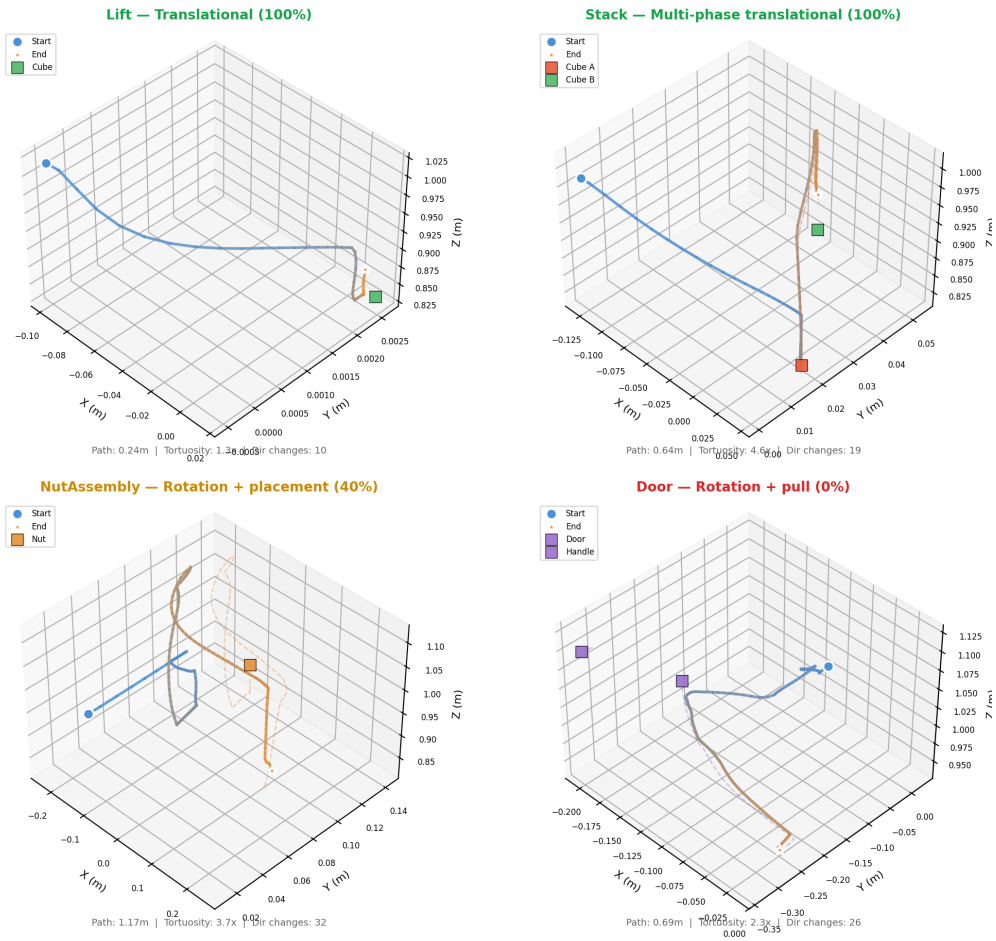


Figure 3: Trajectories Overlook on the 4 Tasks

Table 2: **Iteration trajectory for Stack (0%→100%).** The diagnostic system identifies "eef alignment error" and the agent fixes it in one revision.

Attempt	Success	Diagnosis	Agent Action
1	0% (0/10)	"eef not aligning with cube positions"	—
2	100% (10/10)	— (success)	Fixed approach alignment logic

Table 1: **Success rates across four robosuite tasks of increasing complexity.** Simple manipulation tasks succeed reliably on the first attempt. Contact-rich tasks with precise geometric requirements show variable, low success—reflecting a fundamental limitation of current LLMs, not a pipeline bug.

Task	Geometric Precision	First Attempt	Iteration Helps?	Best Rate
Lift	Low (XYZ translation)	100%	N/A (succeeds immediately)	**100%**
Stack	Medium (two-object)	0–100%	Yes (0%→100%)	**100%**
NutAssembly	High (quaternion math)	0–60% (variable)	No (regresses)	40%
Door	High (rotation + pull)	0%	No (stuck)	0%

Table 1 breaks each task into what the policy must control. Lift and Stack only need position control: move the gripper to a target, close, lift. The LLM gets this right every time. NutAssembly adds two requirements: compute a grasping offset from the nut's orientation, and rotate the gripper to align with the handler before grasping. Both are hard, but each happens once in its own phase. The LLM sometimes gets them right, sometimes not. Hence the 0–60% variance across runs.

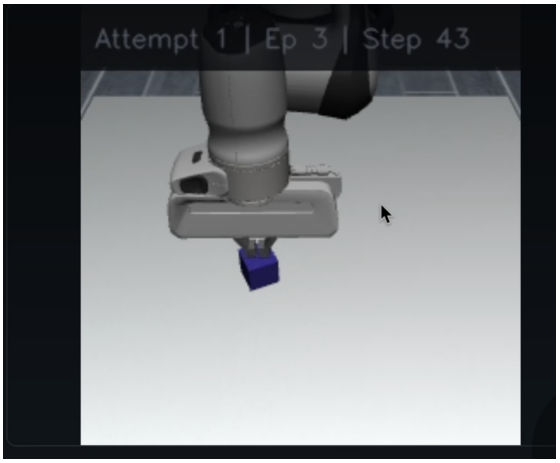
Door looks simpler in the trajectory plot. The end-effector path is shorter and less curved than NutAssembly's. But the path does not show what makes Door hard. To turn a door handle, the policy must do three things at once: hold the gripper on the handle, twist the wrist to rotate the latch, and maintain enough contact to avoid slipping. These three requirements are coupled. If the gripper drifts while twisting, it loses the handle. If it stops twisting to re-center, the latch springs back. The LLM never produces code that coordinates all three.

Why the LLM fails on Door specifically. The LLM generates orientation commands as fixed constants: `action[3:6] = [-0.3, 0, 0]` every step. This is open-loop. The code pushes the same rotation regardless of what the gripper is actually doing. In practice, the gripper drifts off the handle within a few steps. A working policy would read the current orientation, compare it to a target angle, and compute a correction each step. This pattern is standard in robotics but rare in the code the LLM was trained on. Door also has two angle measurements that are easy to confuse: `handle_qpos` measures how far the latch has turned, while `hinge_qpos` measures how far the door has opened. The policy needs the first to know when to stop twisting and start pulling, and the second to check success. The LLM consistently uses the wrong one.

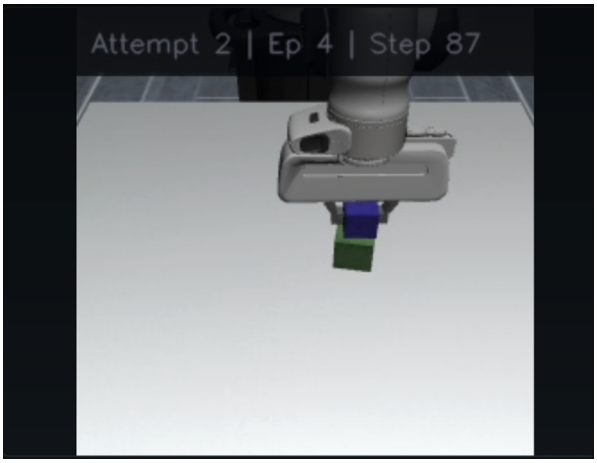
The "lottery" of LLM code generation. Each run with the same prompt produces different code because of sampling randomness. For Lift, almost any plausible code works. There are few ways to get "move to cube, close gripper, lift" wrong. For NutAssembly, several precise computations must all be correct at once: quaternion-to-yaw extraction, handler offset rotation, placement offset. The chance of getting all of them right in one sample is low, so success varies widely across runs. For Door, the required control pattern is not in the LLM's training distribution at all. No amount of re-sampling produces it.

5.2 Iteration Behavior

We find that iterative revision reliably helps for medium tasks (Stack: 0%→100% in one diagnostic-driven revision) but does not improve performance on hard tasks. On NutAssembly, the agent achieves 40% on



(a) Attempt 1 (0%): Misaligned grasp; the end-effector does not descend to the cube center.



(b) Attempt 2 (100%): Corrected alignment; successful stacking across episodes.

Figure 4: **Diagnostic-driven iteration on Stack.** The agent identifies an alignment error from simulation feedback and revises the policy to descend to the object's center before grasping, resulting in reliable task completion.

attempt 2 but regresses to 0% on subsequent attempts—it rewrites the approach logic to fix the diagnosed issue but breaks phases that were already working. On Door, the agent receives the same diagnosis four consecutive times ("eef too far from handle") and cannot fix it. This is consistent with our analysis: iteration is effectively regeneration, not debugging. The LLM does not understand why the code failed, so each revision is a new sample from the same distribution.

6 Discussion

Why iteration helps for simple tasks but not hard ones. LLMs are pattern matchers, not physical reasoners. When the model generates a policy, it produces tokens that look like plausible code given the prompt. It does not reason about physics, geometry, or contact mechanics. For simple tasks like Lift and Stack, the pattern is constrained enough that almost any reasonable code works. There are few ways to get "move to cube, lower, close gripper, lift" wrong. For hard tasks like NutAssembly, several precise things must be correct at once: quaternion-to-yaw extraction, handler offset rotation, placement offset compensation, gripper orientation. The chance of getting all of them right in a single generation is low. Iteration does not reliably help because the model does not understand why the code failed. Each revision is a new sample from the same distribution, not a targeted fix.

This explains the regression on NutAssembly (40%→0%). The LLM sees "approach not targeting handler," rewrites the approach phase, and in doing so breaks parts that were already working. It is not debugging. It is regenerating and hoping for a better draw.

Human-in-the-loop: a design lesson. We designed RoboScribe for human-in-the-loop iteration—a feedback box where the user could guide the agent after each failed test. In practice, we found that human feedback during code iteration does not improve outcomes. The human can see the robot failing ("the gripper missed the handle") but cannot articulate the precise numerical fix that the LLM needs ("change the approach offset from `[0.10, 0.0, 0.05]` to `[0.03, -0.01, 0.0]`"). The gap between what a human can observe and what the code needs is too large for natural language to bridge.

The effective human intervention point turned out to be phase design—defining the strategy before code generation, not debugging code after. A researcher who says "add a hover phase before grasping" changes the

structure of the generated code in a way that actually helps. A researcher who says "the gripper is too far" during iteration does not. This is itself a contribution: it tells practitioners where to invest human time in LLM-driven robotics pipelines.

The remaining high-value intervention is domain constraints—encoding geometric patterns (handler offset formula, pull direction derivation) that the LLM cannot derive from first principles. These constraints are not scaffolding to be removed—they are the system's core value, representing reusable domain expertise that benefits all users.

The path forward. Our findings suggest several directions beyond the current architecture: (1) constrained generation—provide templates with critical math pre-filled, letting the LLM only tune gains and thresholds; (2) code patching—use diff-based revision instead of full policy regeneration to prevent regression; (3) richer few-shot libraries—more working examples increase the probability the LLM copies correct patterns.

Limitations. The system is evaluated with GPT-4o on four environments. Success rates vary across LLM backends—our prompts were initially tuned for Qwen. The iteration loop does not preserve partially-working code when attempting fixes, leading to regressions.

7 Conclusion

We present RoboScribe, a language-guided pipeline for generating robosuite scripted policies from natural language. Our evaluation across four tasks of increasing geometric complexity reveals a clear pattern: LLMs reliably generate working policies for translational manipulation tasks (100% on Lift and Stack) but struggle with tasks requiring precise geometric reasoning (NutAssembly, Door). Critically, the diagnostic system correctly identifies failures in every case—the bottleneck is code synthesis, not failure detection. We also find that human feedback during code iteration does not improve outcomes; the effective intervention point is phase design, where human strategy expertise has the highest leverage.

These findings support a phase-design-first architecture: humans define the strategy, domain constraints encode geometric knowledge, and the agent handles the implementation.

Future investigation. Our results raise a deeper question: is the failure on geometrically complex tasks a fundamental limitation of current LLM architectures, or can it be overcome with better prompting, fine-tuning, or tool-augmented generation? We plan to test this through controlled ablations: (1) providing geometric formulas in the prompt vs. requiring the LLM to derive them; (2) comparing across model families (GPT-4o, Claude, Qwen) to determine if failures are model-specific or class-wide; (3) evaluating constrained generation approaches where the LLM fills parameters in pre-built templates rather than generating full policies.

8 Contributions

Yuer Tang works on choosing topic, infra code strcutrue, agent overall design, leading the overall project. Zhiyi Chen works on the full-stack RoboScribe system, including FastAPI backend, agent runner, and a real-time React interface with live simulation streaming.

References

[1] M. Ahn et al. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. In *CoRL*, 2022.

[2] W. Huang et al. Inner Monologue: Embodied Reasoning through Planning with Language Models. In *CoRL*, 2022.

[3] J. Liang et al. Code as Policies: Language Model Programs for Embodied Control. In *ICRA*, 2023.

[4] W. Huang et al. VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models. In *CoRL*, 2023.

[5] W. Yu et al. Language to Rewards for Robotic Skill Synthesis. In *NeurIPS*, 2023.

[6] Y. J. Ma et al. Eureka: Human-Level Reward Design via Coding Large Language Models. In *ICLR*, 2024.

[7] A. Mandlekar et al. MimicGen: A Data Generation System for Scalable Robot Learning using Human Demonstrations. In *NeurIPS*, 2023.

[8] Y. Li et al. RoboGen: Towards Unleashing Infinite Data for Automated Robot Learning via Generative Simulation. In *ICML*, 2024.

[9] K. Valmeekam et al. On the Planning Abilities of Large Language Models. In *NeurIPS Workshop*, 2023.

[10] G. Wang et al. Voyager: An Open-Ended Embodied Agent with Large Language Models. In *NeurIPS*, 2023.