

A Case Study of On-Premise ReAct for Collaborative Robot Programming

Yu-Seung Ma and Sangyeop Yeo

Electronics and Telecommunications Research Institute (ETRI), Korea

{ysma, sangyeop}@etri.re.kr

Abstract

LLM-based coding agents are increasingly used in software development, but their use in industrial robot programming remains constrained by vendor-specific APIs, organization-specific code assets, and on-premise deployment requirements. We study a bounded ReAct agent that uses local retrieval over API documentation and code examples to generate collaborative-robot programs from natural-language requirements.

This work is a case study of a structured programming domain in which domain knowledge must remain within the organizational boundary and local models may be sufficient. We evaluate seven open-weight models from 7B to 32B parameters, with five independent runs per model, and include one commercial model as a reference. We also apply the same framework to a smaller public Universal Robots set.

Under our agent and evaluation setup, the strongest local 27B model achieves a mean task success rate of 89.6%. Smaller models often use more steps and tokens because they repeat retrieval or fail to convert retrieved evidence into code. These results suggest that model selection should consider task success, interaction efficiency, and run-to-run consistency.

1 Introduction

LLM-based coding agents increasingly support software development from natural-language requirements (Jimenez et al., 2024), but industrial robot programming presents a different setting. Robot software often depends on vendor-specific APIs, examples, and execution conventions that may be proprietary and poorly represented in pre-training data. Industrial deployment may also require on-premise inference, limiting the use of external services and large hosted models.

This work examines whether a standard bounded ReAct agent (Yao et al., 2023), combined with local retrieval, is useful in such an environment. The

agent searches a local knowledge base of API documentation and code examples and generates programs from natural-language requirements. The target domain is relatively structured: collaborative-robot programs often follow recurring patterns of motion, I/O, communication, and parameter handling. This makes it a suitable case for testing whether smaller local models can use retrieved domain knowledge effectively.

We evaluate seven open-weight models from 7B to 32B parameters on 150 industrial tasks derived from programs used by an industry partner and on 54 public Universal Robots tasks. Each open-weight model is evaluated in five independent runs, and Claude Sonnet is included as a commercial reference. We evaluate both task success and agent trajectories across 7,344 retrieval-enabled episodes.

Our contributions are as follows:

- We study a bounded on-premise ReAct agent with local retrieval for collaborative-robot programming using vendor-specific documentation and organization-specific code examples.
- We evaluate seven open-weight models from 7B to 32B parameters on 150 industrial tasks over five independent runs, considering both final correctness and process-level behavior, including retrieval, no-progress turns, repair, cost, and run-to-run consistency.
- We conduct a limited cross-platform evaluation on 54 public URScript tasks by replacing the platform-specific knowledge sources and evaluator, and report a single-run commercial model as a descriptive external reference.

2 Background and Related Work

Industrial robot programming relies on vendor-specific languages, APIs, and execution conventions that may be poorly represented in model pre-training. In on-premise deployments, such knowledge must also remain within the organizational

boundary and be supplied at inference time. Prior work reports degraded performance on structurally modified ABB RAPID programs (Fares and Herbold, 2025) and substantial gains from retrieval for legacy PLC languages (Yeh et al., 2026), although PLC and robot programming involve different control abstractions.

LLM-based robot programming has progressed from policy generation over predefined APIs (Liang et al., 2023; Singh et al., 2023) to ROS2-based programming, low-code interfaces, and multi-robot planning (Dong et al., 2025; Su et al., 2026). Most studies assume public or fixed platform interfaces. More closely related work uses vendor-owned assets for secure on-premise code generation (Kersting et al., 2026), retrieval for robot code generation (Salunkhe et al., 2026), or fine-tuning for URScript (Le and Le, 2025). In contrast, we maintain platform knowledge in a replaceable local index that can be updated without retraining.

Our agent follows ReAct (Yao et al., 2023) and uses retrieval-augmented generation (Lewis et al., 2020) as an optional in-loop action. Building on recent work on tool use and agent-trajectory analysis (Lu et al., 2025; Ou et al., 2025; Cao et al., 2026), we evaluate not only final correctness but also retrieval behavior, unproductive actions, repair, repeated-run dependability, and end-to-end cost. Our contribution is a controlled process-level evaluation of a bounded on-premise robot-programming agent, rather than a new agent architecture.

3 Bounded ReAct Agent

We study a bounded ReAct agent for collaborative-robot programming under an on-premise deployment setting. The purpose is not to propose a new agent architecture, but to examine how local language models retrieve and apply platform-specific knowledge within a fixed agent framework.

3.1 Tasks and Knowledge Sources

Our primary industrial evaluation set contains 150 natural-language programming tasks prepared with engineers from an industry partner. The tasks are derived from real-world robot programs and encompass motion control, loops, I/O control, communication, and vendor-specific parameter handling.

We construct a local retrieval index of API descriptions and worked instruction-program examples using dense retrieval with top- k results. These

Instruction (translated). “Monitor PLC01, which sends one message per line. When a message arrives, turn on tool digital output 2 if general digital input 1 is active.”

Knowledge obtained by retrieval.

```
setSocketDelimiters(name, prefix, suffix);
addSocketListener(name, event, cb);
digital-input and tool-output signatures;
and a worked callback example from the local corpus.
```

Reference behavior (executable).

```
var SOCKET_NAME = 'PLC01';
setSocketDelimiters(SOCKET_NAME, ' ', '\n');
addSocketListener(SOCKET_NAME, 'data',
function (d) {
  if (readDigitalInput(1) == 1)
    writeToolOutput(2, 1);
});
```

Figure 1: A simple illustrative industrial task. Solving this example requires retrieving the relevant socket and I/O APIs and preserving the specified event, delimiter, input condition, and output behavior.

materials are available only through retrieval during the agent loop, representing deployments in which organizational assets must remain within the organizational boundary.

For each task, the corresponding worked example is excluded from the retrieval index to prevent direct access to the reference solution. Related examples, however, are intentionally retained. In the target deployment scenario, the agent is expected to retrieve and adapt existing programs that encode vendor-specific APIs and organization-specific programming conventions. Similarity between a task and retrieved examples is therefore an intended property of the evaluation setting. The industrial evaluation measures adaptation using organizational memory, rather than generalization to fully disjoint program families.

We additionally evaluate the framework on 54 tasks from the public URScript platform. This smaller set is based on a publicly available API and serves as a limited cross-platform check. Only the platform-specific documentation, worked examples, and evaluator are replaced; the agent loop and interaction procedure remain unchanged. The URScript set is not intended as a second full-scale benchmark. Figure 1 illustrates the task format by linking an instruction, retrieved knowledge, and executable code.

3.2 Agent Loop and Validation

Figure 2 summarizes the agent workflow. Before the ReAct loop, a planner converts the instruction into a checklist of required behaviors. The checklist helps the agent track the task requirements but does

not provide vendor-specific API knowledge.

At each turn, the model either issues `search_rag [query]` command or emits a candidate program. Retrieval returns relevant API descriptions or code examples, and the model decides whether the available evidence is sufficient for code generation.

The ReAct loop is limited to eight model turns. Each generated program is checked by a deterministic validator for syntax, allowed API usage, function signatures, and behavior under mock execution. If validation fails, the diagnostic message is returned to the model for up to four repair turns.

The final outcome is labeled PASS when the program matches the required behavior, PARTIAL when a program is produced but is behaviorally incorrect or uses an invalid API, and FAIL when no program is produced within the interaction budget. This setup allows us to test whether a simple on-premise ReAct agent can retrieve vendor-specific knowledge and convert it into valid collaborative-robot programs. Appendix A presents one complete agent trace.

4 Evaluation Design

We evaluate the agent from three perspectives: final task success, the efficiency of the generation process, and consistency across repeated runs. We also compare it with a commercial model and test the same framework on a second robot platform.

Metrics. An *episode* is one execution of a task, and each model response is an *agent step*. From the logs, we identify productive retrievals, repeated retrievals, no-progress steps, code emission, and repair. Repeated retrieval is classified as exact or semantic, and a no-progress step produces neither a valid tool command nor a program.

Let S_i be the number of steps in episode i , with D_i exact repeated retrievals, N_i no-progress steps, and E_i repeated validation errors. Let R_i^{prod} and R_i denote productive and total retrievals:

$$\text{WSR} = \frac{\sum_i (D_i + N_i + E_i)}{\sum_i S_i}, \quad \text{PRR} = \frac{\sum_i R_i^{\text{prod}}}{\sum_i R_i}. \quad (1)$$

The wasted-step rate (WSR) measures unproductive interaction, while the productive-retrieval rate (PRR) measures retrieval utility. We also report retrieve-to-commit, the number of steps from the last retrieval to code emission.

Each open-weight model is evaluated on task set

$\mathcal{T}$ in $r = 5$ independent runs. Following pass^k -style reliability evaluation (Yao et al., 2025), we report the all-five-run pass rate as the stable pass rate (SPR):

$$\text{SPR} = \frac{1}{|\mathcal{T}|} \left| \{t \in \mathcal{T} : \text{pass}_j(t) = 1 \ \forall j \leq r\} \right|. \quad (2)$$

The gap between the mean pass rate and SPR summarizes the difference between average task success and all-five-run task coverage.

Models and runs. Table 1 lists the evaluated models. We abbreviate DeepSeek-R1-Distill-Qwen-32B as R1-Distill-32B in subsequent tables and figures. The seven open-weight models are served on premise through vLLM (Kwon et al., 2023) in BF16 and evaluated in five full runs under both retrieval-enabled and no-retrieval conditions, without seed pinning (temperature 0.2, top- $p = 0.95$). The ablation disables only the retrieval tool, leaving the prompt, planner, interaction budget, and validator unchanged.

Table 1: Evaluated models and serving conditions.

Deployment	Model
On premise (vLLM/BF16)	Qwen2.5-Coder-7B
	Qwen2.5-Coder-14B
	Qwen2.5-Coder-32B
	Codestral-22B
	Gemma-3-27B-IT
	DeepSeek-R1-Distill-Qwen-32B
	Qwen3.6-27B
Commercial	Claude Sonnet 5 [†]

[†] Accessed through Claude Code; model revision and serving environment are provider-managed.

Claude Sonnet is evaluated once under the retrieval-enabled setting as an external reference. Token and latency comparisons exclude Claude because its serving environment is not controlled.

5 Results

5.1 RQ1: Behavioral correctness

Table 2 reports the retrieval-enabled results on the 150 industrial tasks, with the outcome distribution shown in Figure 3. In a separate no-knowledge ablation, all evaluated models achieved 0/150 PASS. Under our setup, this result indicates that the tasks required access to vendor-specific knowledge.

With retrieval, performance improves substantially across models. Within the Qwen2.5-Coder family, mean PASS outcomes increase from 12.2 at 7B to 91.4 at 14B and 130.6 at 32B. Codestral-22B, Gemma-3-27B-IT, and R1-Distill-Qwen-32B

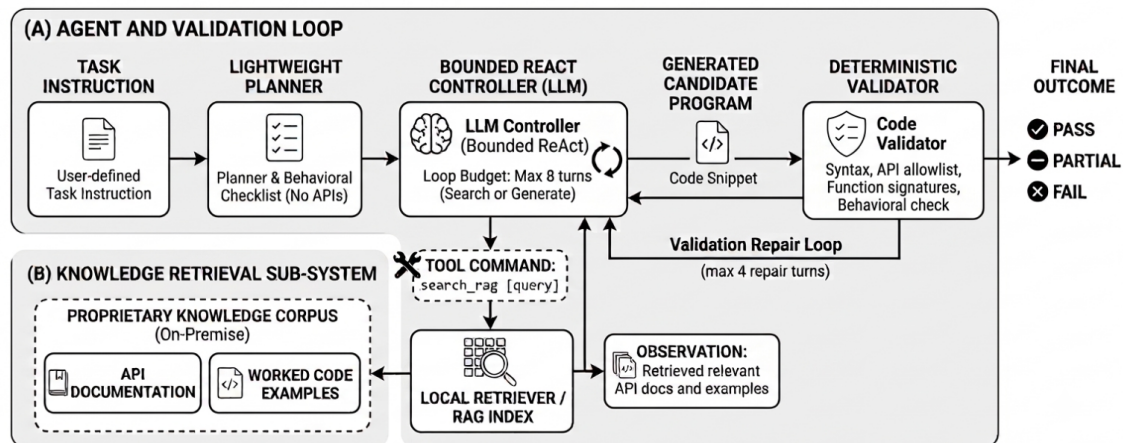


Figure 2: Instrumented on-premise coding-agent workflow. Retrieval is invoked from within the bounded ReAct loop as a tool command followed by an observation. The same agent loop is used with different platform-specific knowledge assets, and measurements are collected at the retrieval, interaction, and program-validation stages.

Table 2: Behavioral outcomes on the 150-task industrial set. All open-weight models report means over five independent runs; Claude Sonnet is a single-run commercial reference.

Model	PASS	PART.	FAIL	PASS%
Qwen2.5-Coder-7B	12.2	108.8	29.0	8.1
Qwen2.5-Coder-14B	91.4	53.8	4.8	60.9
Qwen2.5-Coder-32B	130.6	19.4	0.0	87.1
Codestral-22B	96.8	47.4	5.8	64.5
Gemma-3-27B-IT	66.2	64.0	19.8	44.1
R1-Distill-32B	91.2	47.0	11.8	60.8
Qwen3.6-27B	134.4	13.6	2.0	89.6
Claude Sonnet 5	135	15	0	90.0

Table 3: Task-level flakiness over five runs. Columns 0–5 show the number of tasks passing in that many runs; SPR is the all-five-run pass rate.

Model	0	1	2	3	4	5	SPR
Coder-7B	108	29	9	2	2	0	0.0
Coder-14B	13	14	24	32	36	31	20.7
Coder-32B	3	7	8	4	22	106	70.7
Codestral-22B	13	12	18	32	35	40	26.7
Gemma-3-27B-IT	34	31	22	19	21	23	15.3
R1-Distill-32B	16	15	22	25	38	34	22.7
Qwen3.6-27B	4	4	2	9	18	113	75.3

achieve 96.8, 66.2, and 91.2 mean PASS outcomes, respectively. Qwen3.6-27B gives the best on-premise result at 134.4/150 (89.6%), while the single-run Claude Sonnet reference obtains 135/150 (90.0%). The commercial comparison is descriptive. Qwen2.5-Coder-32B produces no FAIL outcomes but still averages 19.4 PARTIAL outputs per run.

5.2 RQ2: Dependability across repeated runs

Table 3 and Figure 4 show that task outcomes vary across repeated runs. Some tasks pass in some runs but not in others. We refer to tasks that pass in

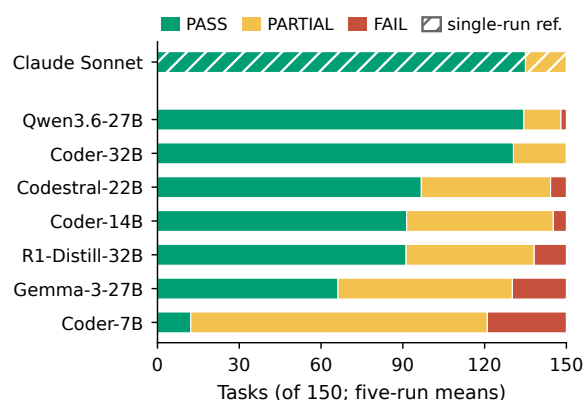


Figure 3: Outcome composition over 150 industrial tasks. Open-weight rows are five-run means; the hatched row is the single-run external commercial reference.

at least one but fewer than all five runs as *flaky* tasks. The mid-range models have 93–106 such tasks, and their mean-SPR gaps range from 28.8 to 40.2 percentage points (Figure 5). This indicates that mean pass rate and all-five-run task coverage provide different views of model performance.

The type of variation also differs across models. For Qwen2.5-Coder-32B, all 41 flaky tasks alternate between PASS and PARTIAL, whereas weaker models more often include FAIL outcomes across runs. Fleiss’ κ ranges from 0.09 for Qwen2.5-Coder-7B to 0.44 for Qwen2.5-Coder-32B (Table 4), indicating different levels of agreement across repeated evaluations.

Process behavior also varies across runs. WSR varies by 1.4–3.4 percentage points, and only 11–33% of always-pass tasks use the same number of ReAct turns in all five runs (Table 4). Thus, outcome consistency and trajectory consistency need not coincide.

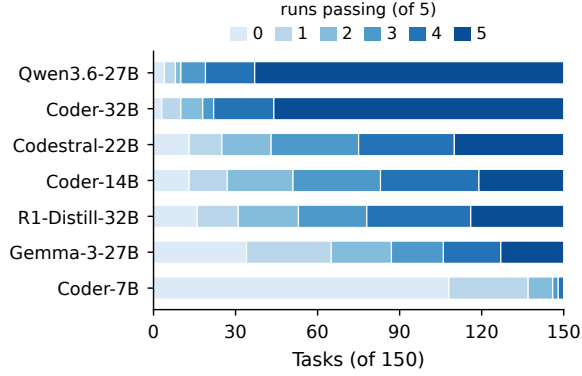


Figure 4: Distribution of the 150 industrial tasks by number of passing runs (darker = more of five runs). The ends of the capability ladder are nearly bimodal; the middle of the ladder is dominated by flaky tasks.

Table 4: Re-run consistency over five runs. Fleiss’ κ measures agreement among the three-way verdicts. Flaky tasks are divided into PASS $\leftrightarrow$ PARTIAL flips and flips involving FAIL. Same traj. is the share of always-pass tasks with identical ReAct turn counts across runs.

Model	κ	P $\leftrightarrow$ Pt	w/ FAIL	Same traj.
Coder-7B	0.09	21	21	—
Coder-14B	0.23	86	20	23%
Coder-32B	0.44	41	0	33%
Codestral-22B	0.27	77	20	30%
Gemma-3-27B-IT	0.36	53	40	26%
R1-Distill-32B	0.26	67	33	12%
Qwen3.6-27B	0.43	24	9	11%

An earlier BF16 evaluation of Qwen2.5-Coder-7B on a different runtime produced 21 passes, outside the range observed with vLLM/BF16. This result suggests that small-model evaluations may also depend on the serving environment. For this reason, repeated evaluations are best compared under a common serving setup.

5.3 RQ3: Where the budget goes

Table 5 summarizes the interaction budget. Within the Qwen2.5-Coder family, the mean number of steps decreases from 3.6 at 7B to 2.7 at 32B, while tokens decrease from 12.0k to 6.6k per task. Over the same scale, WSR falls from 21.8% to 3.0%, and PRR rises from 67.7% to 98.7%. Larger models in this family therefore use fewer steps and make better use of retrieval.

Retrieve-to-commit remains close to 1.0–1.3 steps across models, indicating that code is generally emitted soon after the final retrieval. Qwen3.6-27B completes 47% of tasks within two steps, and repair is invoked in only 14 of its 750 episodes. These results suggest that effective models use retrieval mainly for targeted grounding rather than extended exploration.

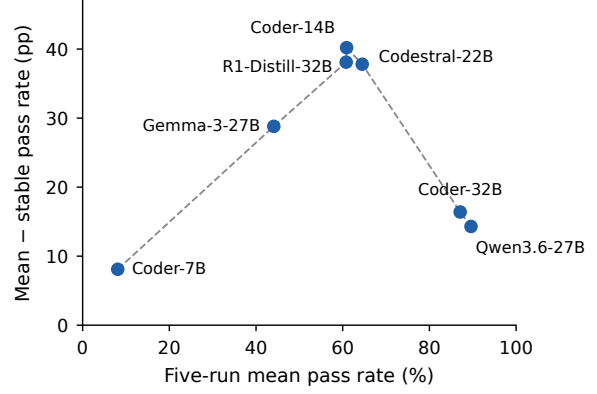


Figure 5: Gap between the five-run mean pass rate and stable pass rate. The gap is largest for the mid-range models (28.8–40.2 percentage points), indicating greater task-level variability across runs.

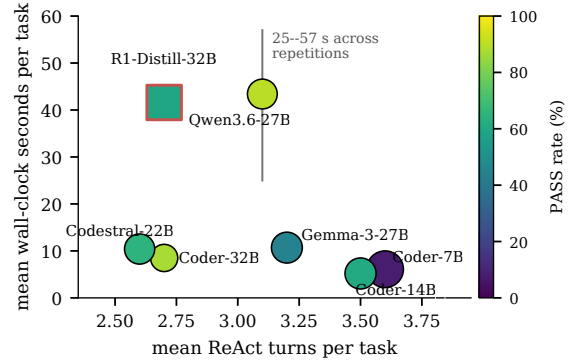


Figure 6: Interaction-cost trade-off (on-premise sweep). Marker area $\propto$ tokens per task; color = PASS; all points are five-run means on the shared vLLM/BF16 stack. Sonnet excluded (uncontrolled serving stack).

5.4 RQ4: Repetition without progress

Table 6 separates repeated retrieval, no-progress behavior, and repeated validation errors. The 7B model averages 0.43 exact retrieval repetitions and 0.33 no-progress steps per episode. For example, in one logged episode, the first retrieval returns a worked example that nearly solves the task, but the model continues searching and exhausts the eight-step budget without emitting code. This suggests that the failure arises not from missing evidence, but from failing to recognize that the retrieved evidence is sufficient.

Gemma-3-27B-IT produces many no-progress steps because prose makes its tool commands unparsable. R1-Distill-Qwen-32B shows the same pattern less often. Repeated retrieval reflects poor evidence use, whereas no-progress behavior reflects failure to follow the tool interface.

Table 5: Process statistics. Open-weight results aggregate five runs ($150 \times 5 = 750$ episodes per model); Claude Sonnet is a single-run commercial reference, with provider-managed token and latency statistics omitted. WSR and PRR are defined in Eq. 1. Retrieve-to-commit is the mean number of steps from the final retrieval to code emission over code-emitting episodes.

Model	Steps	Tokens	Sec.	WSR	PRR	Retrieve-to-commit
Qwen2.5-Coder-7B	3.6	12.0k	6.1	21.8%	67.7%	1.20
Qwen2.5-Coder-14B	3.5	8.6k	5.2	13.9%	90.3%	1.00
Qwen2.5-Coder-32B	2.7	6.6k	8.5	3.0%	98.7%	1.00
Codestral-22B	2.6	8.1k	10.4	4.1%	99.0%	1.00
Gemma-3-27B-IT	3.2	8.5k	10.7	33.3%	98.9%	1.26
R1-Distill-32B	3.3	9.6k	42.9	13.6%	98.9%	1.00
Qwen3.6-27B	3.1	7.9k	43.4	6.9%	91.4%	1.17
Claude Sonnet 5	2.6	–	–	9.6%	81.9%	1.25

Table 6: Trajectory behaviors observed in the campaign logs. Open-weight model rows aggregate five runs ($150 \times 5 = 750$ episodes per model); Claude Sonnet is evaluated in a single run (150 episodes). Each cell reports the raw count and the episode-normalized rate in parentheses.

Model	Exact repeated retrieval	Semantic repeated retrieval	No-progress turns	Repeated validation error
Qwen2.5-Coder-7B	323 (0.43)	59 (0.08)	250 (0.33)	21 (0.03)
Qwen2.5-Coder-14B	358 (0.48)	48 (0.06)	2 (0.00)	0 (0.00)
Qwen2.5-Coder-32B	59 (0.08)	11 (0.01)	0 (0.00)	0 (0.00)
Codestral-22B	25 (0.03)	20 (0.03)	56 (0.07)	0 (0.00)
Gemma-3-27B-IT	3 (0.00)	17 (0.02)	779 (1.04)	21 (0.03)
R1-Distill-32B	91 (0.12)	1 (0.00)	248 (0.33)	0 (0.00)
Qwen3.6-27B	73 (0.10)	84 (0.11)	90 (0.12)	0 (0.00)
Claude Sonnet 5	0 (0.00)	0 (0.00)	38 (0.25)	0 (0.00)

5.5 RQ5: Cost–quality trade-off

Figure 6 shows that a smaller backbone does not necessarily produce a more efficient agent. The 7B model has the lowest pass rate while using the most steps and tokens per task. Model selection should therefore consider task success, interaction length, token use, and latency together. Qwen3.6-27B varies from 25 to 57 seconds per task across runs despite similar interaction patterns, illustrating the effect of serving conditions on wall-clock time.

5.6 RQ6: Cross-platform check

Our primary evaluation uses tasks provided by a single industrial partner. To assess whether the same framework also applies to another robot platform, we additionally evaluate it on 54 public URScript tasks, replacing only the platform-specific API documentation and code examples. The agent, retrieval mechanism, prompts, and evaluation procedure remain unchanged. Task success is assessed from the required API-level behavior, with simulator inspection for motion-related tasks.

Table 7 summarizes the results. Qwen2.5-Coder-32B achieves the highest open-weight result at 51.8/54 tasks on average, while Claude Sonnet 5 completes 53/54 tasks in a single reference run. These results show that the same bounded

Table 7: URScript results on the 54-task set for all evaluated models. Open-weight values are five-run means; Claude Sonnet 5 is a single-run reference.

Model	Task success, count (%)
Qwen2.5-Coder-7B	21.0 ± 4.5 (38.9%)
Qwen2.5-Coder-14B	45.4 ± 1.5 (84.1%)
Qwen2.5-Coder-32B	51.8 ± 0.8 (95.9%)
Codestral-22B	41.2 ± 0.8 (76.3%)
Gemma-3-27B-IT	44.4 ± 0.9 (82.2%)
R1-Distill-32B	44.0 ± 2.3 (81.5%)
Qwen3.6-27B	44.4 ± 1.5 (82.2%)
Claude Sonnet 5	53 (98.1%)

ReAct framework can also be applied to a second collaborative-robot platform by replacing the platform-specific knowledge source. The observed results may partly reflect the availability of public URScript documentation and examples. The smaller URScript evaluation therefore serves as an additional platform check rather than evidence of broad generalization.

6 Discussion

Model selection should consider not only task success but also run-to-run consistency and process efficiency. A smaller model may be cheaper per call yet use more interaction if it repeats retrieval, produces no-progress steps, or exhausts the budget

without generating code. The results also show that successful retrieval is not sufficient: the model must recognize when the retrieved evidence is adequate and produce a valid program. These behaviors should therefore be reported separately from final correctness.

The stable pass rate provides a practical estimate of dependable task coverage, while flaky tasks indicate the likely review burden. For deployment, evaluations should report repeated runs alongside the mean pass rate, distinguish repeated retrieval from tool-interface failures, and compare models under a common serving setup. A bounded agent loop, API validation, and mock execution should remain in place before hardware execution.

7 Conclusion

We presented a process-level case study of a bounded on-premise ReAct agent for collaborative-robot programming. Across 7,344 retrieval-enabled episodes and five repeated runs per open-weight model, local models were able to retrieve and apply platform-specific documentation and organization-specific code examples within a fixed agent framework. Models with higher task success generally followed shorter and more productive trajectories, whereas weaker models more often repeated retrieval or produced no-progress turns.

The repeated evaluations also showed that mean task success may overstate dependable task coverage, particularly for models of moderate capability. The limited URScript evaluation suggests that the same framework can be applied to another robot platform by replacing the platform-specific knowledge sources and evaluator, although it does not establish broad cross-platform generalization. Overall, the results indicate that on-premise model selection should consider final correctness, run-to-run consistency, and interaction efficiency together.

Limitations

The industrial evaluation covers one robot platform, one programming language, and tasks from a single partner, so the results may not generalize to other settings. Although each task’s own example is excluded from retrieval, related programs remain in the repository; the evaluation therefore measures adaptation with organizational memory rather than generalization to unseen program families.

The 54-task URScript evaluation is only a limited cross-platform check, and API-sequence agree-

ment may miss argument or data-flow errors. Claude Sonnet is evaluated once on a provider-managed stack, so the comparison is descriptive. Mock execution also assesses API-level behavior rather than physical safety, which requires additional validation and human review.

Ethical Considerations

The primary dataset contains proprietary industrial instructions and no personal data. Generated programs are evaluated through static checks, mock execution, and simulation rather than direct execution on a production robot; behavioral equivalence does not imply physical safety, and any deployment should retain vendor-certified runtime protections, workspace constraints, and human review.

Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2022-0-00995, Automated reliable source code generation from natural language descriptions)

References

- Hongliu Cao, Ilias Driouich, and Eoin Thomas. 2026. [Beyond task completion: Revealing corrupt success in LLM agents through procedure-aware evaluation.](#) *Preprint*, arXiv:2603.03116.
- Wenhang Dong, Dongpeng Li, Yuchen Ji, Hongpeng Chen, Shimin Liu, Zheng Ma, Fang Hao, Yuqi Ji, Hongwen Xing, and Pai Zheng. 2025. [Towards a next-generation LLM empowered low-code programming industrial robotic system for human-centric smart manufacturing.](#) *Journal of Manufacturing Systems*.
- Salim Fares and Steffen Herbold. 2025. [Utilizing LLMs for industrial process automation: A case study on modifying RAPID programs.](#) *Preprint*, arXiv:2511.11125.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*.
- Joschka Kersting, Michael Rummel, and Gesa Benndorf. 2026. Vendor-aware industrial agents: Rag-enhanced llms for secure on-premise plc code generation. In *2026 IEEE International Conference on Industrial Technology (ICIT)*, pages 1–7. IEEE.

- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.
- Tien Dat Le and Minhhuy Le. 2025. [Empowering universal robot programming with fine-tuned large language models](#). *EAI Endorsed Transactions on AI and Robotics*, 4(1).
- Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control. In *IEEE International Conference on Robotics and Automation (ICRA)*.
- Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. 2025. [ToolSandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities](#). In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1160–1183, Albuquerque, New Mexico. Association for Computational Linguistics.
- Tianyue Ou, Wanyao Guo, Apurva Gandhi, Graham Neubig, and Xiang Yue. 2025. [AgentDiagnose: An open toolkit for diagnosing LLM agent trajectories](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 207–215, Suzhou, China. Association for Computational Linguistics.
- Omkar Salunkhe, Siyuan Chen, Anna Syberfeldt, and Johan Stahre. 2026. [Developing RAGs for robot code generation](#). *IOP Conference Series: Materials Science and Engineering*, 1342(1):012064.
- Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2023. ProgPrompt: Generating situated robot task plans using large language models. In *IEEE International Conference on Robotics and Automation (ICRA)*.
- Xiangyu Su, Juzhan Xu, Oliver van Kaick, Kai Xu, and Ruizhen Hu. 2026. [IMR-LLM: Industrial multi-robot task planning and program generation using large language models](#). *Preprint*, arXiv:2603.02669.
- Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. 2025. [\$\tau\$ -bench: A benchmark for tool-agent-user interaction in real-world domains](#). In *The Thirteenth International Conference on Learning Representations*.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.
- Ming-Feng Yeh, Ching-Chuan Luo, and Cheng-Lin Lu. 2026. [Multi-hardware benchmarking of open-source large language models with retrieval-augmented generation for Mitsubishi FX-series PLC instruction list code generation](#). *Sensors*, 26(11):3602.

A An Agent Trace on the URScript Set

Figure 7 presents one successful episode from the URScript evaluation. All agent turns are retained. The instruction and natural-language portions of the trajectory are translated and lightly edited for readability. The generated program is typeset with formatting-only changes.

Instruction (translated). “Move to a joint target supplied at execution time, wait 500 ms, and then turn on tool digital output 1.”

Planner checklist (pre-loop artifact). joint target provided as a runtime parameter — joint-move execution — 500 ms delay after the move — tool digital output 1 set to True.

ReAct trajectory (translated and lightly edited).

- **Turn 1 — no-progress.** The model embeds its intended command inside the reasoning text. The harness detects no well-formed command and logs a no-progress turn, which contributes to WSR.
- **Turn 2 — retrieval.** search_rag [set_tool_digital_out signature]. The returned catalog entry specifies set_tool_digital_out(n, b) -> void, which sets the level of a tool digital-output pin.
- **Turn 3 — retrieval.** search_rag [movej default parameters]. The returned catalog entry specifies movej(q, a?, v?) -> void, a joint-space move to the supplied target position.
- **Turn 4 — commit.** After verifying the required signatures, the model converts 500 ms to sleep(0.5) and emits a function that receives the joint target at execution time.

Generated program.

```
def execute_task(target_q):
    movej(target_q, a=1.4, v=1.05)
    sleep(0.5)
    set_tool_digital_out(1, True)
end
```

Outcome. The program matches the reference API sequence movej → sleep → set_tool_digital_out, and is therefore counted as an API-sequence match in the URScript evaluation. The program preserves the deferred joint target as the runtime-supplied parameter target_q.

Figure 7: One successful URScript episode showing planner output, a no-progress turn, targeted API retrieval, and a runtime-supplied joint target.