Benchmarking AI Agents for Hardware Design Automation via MCP Tool Calling

Leonardo Liparulo
Politecnico di Milano
leolipa02@gmail.com

Francesco Pierri
Politecnico di Milano
francesco.pierri@polimi.it

Abstract

We ask whether AI agents powered by locally deployed large language models can reliably automate expert-defined hardware design workflows in an industry-realistic tool-calling setting. In these environments, engineers issue repetitive, dependency-ordered operations—such as creating components, adding ports, and wiring connections—through specialised tools. Confidentiality constraints on component specifications and naming conventions often preclude hosted proprietary APIs, motivating the use of locally deployed models. To study this setting, we build a Model Context Protocol (MCP) server that reproduces the state and dependency logic of a proprietary hardware design tool used in embedded system development and construct a benchmark covering single-operation edits, multi-step dependency chains, invalid requests, misspelled prompts, and multi-server tool contexts. We evaluate seven open-source models comparing pipeline choices including system prompts, tool-description detail, context scope, and single-agent versus multi-agent architectures. Results show that strong models can achieve near-complete expected-call coverage on the benchmarked workflows, but reliability depends strongly on both task structure and agent configuration. Comprehensive tool descriptions consistently reduce failures, few-shot prompting can cause severe inaction for some models, cumulative context harms constrained models, and multi-agent decomposition helps weak workers or long sessions at the cost of additional calls. These findings provide practical guidance for deploying local LLM agents in stateful hardware design environments.

1 Introduction

Hardware design for embedded systems is carried out through specialised tools such as Synopsys Virtualizer or the open-source Kactus2 (Kamppi et al., 2012), which expose graphical interfaces for building structured component models. For instance, an engineer creates a component identified by a VLNV (Vendor, Library, Name, Version) tuple, adds signal and transactional ports, defines parameters, instantiates subcomponents, and wires connections between them. These operations must be issued in dependency order: a port must exist before it can be connected, and a subcomponent must exist before its ports can be referenced (see Fig. 1).

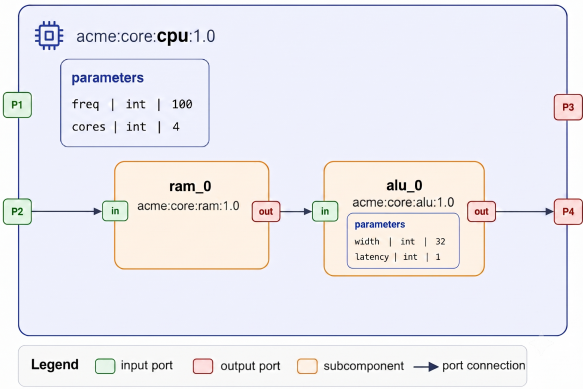

Figure 1: A component in the target application domain. Subcomponents, ports, and connections form a dependency-ordered structure: each element must be created before it can be referenced by subsequent operations.

In industrial practice, many of these interactions are repetitive and template-like, such as adding batches of similarly typed ports, instantiating analogous subcomponents, or wiring connections that follow a regular pattern. Automating them is attractive because it reduces repetitive manual interaction while preserving the structured operations and dependency constraints already enforced by the design tool.

More generally, this kind of workflow consists of structured operations issued in dependency order against persistent application state. Such settings are well-suited to LLM-based tool-calling agents,

which translate natural-language requests into sequences of API calls. Agents operating in this manner have been studied in retail and airline customer operations (Yao et al., 2024), CRM systems (Huang et al., 2025), enterprise workflows (Drouin et al., 2024), and multi-application environments (Trivedi et al., 2024). In these settings, the agent modifies shared state through structured tool calls, and later operations depend on the results of earlier ones. The Model Context Protocol (MCP) (Anthropic, 2024a) standardises this style of agent–application interaction and has seen rapid adoption as an integration layer.

Hardware design shares the same stateful and dependency-ordered characteristics as these agent-driven domains, but also introduces practical deployment constraints. Component specifications, port names, and internal naming conventions can reveal details of unreleased products, often precluding the use of hosted proprietary APIs. Locally deployed open-source models avoid this exposure, but may be less capable than frontier-hosted models. This raises a practical question for industrial adoption: **can LLM-based agents reliably automate expert-defined hardware design workflows in an industry-realistic tool-calling setting while operating within these constraints?**

We address this question by implementing an MCP server whose 14 tools cover the recurrent component-editing operations and dependency logic identified with professional users of a proprietary tool. The server reproduces the relevant data model and constraints of the tool rather than calling its production APIs directly. We then evaluate agents built on seven open-source models running locally through Ollama (Ollama, 2024) as 4-bit quantised variants.

Our contributions are: (1) an MCP server for the proprietary hardware design tool; (2) an expert-informed benchmark comprising eight task suites, spanning independent tasks, dependency chains, cross-task sessions, error handling, and multi-server contexts; (3) a systematic evaluation across prompt, tool-description, context-management, and architecture choices; and (4) practitioner guidelines for deploying reliable local agents in hardware design environments.

2 Related Work

Agents that operate applications. LLM agents that operate applications through stateful tool calls are evaluated by benchmarks such as τ-bench (Yao et al., 2024), CRMArena (Huang et al., 2025), WorkArena (Drouin et al., 2024), AppWorld (Trivedi et al., 2024), and ToolSandbox (Lu et al., 2025). These settings require dependency-ordered interactions over shared mutable state, structurally similar to hardware design workflows.

Because later actions depend on earlier ones, task-level success alone can hide omitted or extraneous calls, motivating call-level evaluation (Yao et al., 2024; Gao et al., 2025). MCP-native frameworks such as MCP-RADAR (Gao et al., 2025) and MCP-Bench (Wang et al., 2025) adopt call-level scoring as a key evaluation strategy for MCP-based tool-calling systems.

LLMs have separately been applied to hardware tasks such as RTL generation (Chang et al., 2023) and EDA-flow orchestration (Fu et al., 2023), typically producing design artefacts or pipeline commands rather than operating a stateful design application through structured tool calls. To our knowledge, prior work does not evaluate MCP-style agents for hardware design workflows involving persistent cross-task state, invalid requests, or multi-server tool contexts.

Agent configuration. Tool-calling performance depends not only on the model but also on the surrounding system design. Orchestration strategy matters: AgentArch (Bogavelli et al., 2025) shows that architectural preferences are model-dependent, with different designs benefiting different model scales. Prompt structure significantly affects tool-use behaviour, with structured and role-based prompts improving compliance (He et al., 2024; Zhang et al., 2024). Tool-description quality shapes tool selection and argument accuracy (Anthropic, 2024b), while long contexts can degrade performance over extended sessions (Liu et al., 2024; Levy et al., 2024). Although these factors have been studied independently, their combined effect in stateful, dependency-ordered environments remains underexplored.

3 Experimental Setup

3.1 MCP Server

We implement a Model Context Protocol (MCP) server that reproduces the state, data model, and dependency constraints of a proprietary hardware design tool through 14 callable tools (Table 1), where later calls may depend on earlier ones.

Group	Tools
Adder	`create_component`, `add_signal_port`, `add_transactional_port`, `add_user_parameter`, `add_generator_parameter`, `add_subcomponent`, `add_subcomponent_connection`
Getter	`get_component_details`, `get_element`, `list_components`
Other	`delete_element`, `delete_component`, `save_component`, `load_component`

Table 1: MCP server tool set.

3.2 Models

We evaluate seven open-source models, all running locally through Ollama (Ollama, 2024) as 4-bit quantised variants: Llama 3.1 8B (Grattafiori et al., 2024), Gemma 4 E4B, Gemma 4 26B, Gemma 4 31B (Google DeepMind, 2026), Qwen 3.5 27B (Qwen Team, Alibaba Cloud, 2026a), Qwen 3.6 27B (Qwen Team, Alibaba Cloud, 2026b), and GPT-OSS 20B (Agarwal et al., 2025). All runs use each model's default Ollama sampling temperature reflecting an out-of-the-box local deployment setting; Section 4.3 reports sensitivity checks at lower temperatures.

3.3 Agent Design

Agent architecture. We compare two architectures: **ReAct** (Yao et al., 2022), a single-agent loop issuing one tool call per turn, and **Plan-and-Act** (Erdogan et al., 2025), a multi-agent design in which a planner decomposes the request, a validator checks the plan, and independent ReAct workers execute the steps (sequence diagrams in Appendix A).

System prompt. Four ReAct variants are evaluated: **none** (tool schemas only), **basic** (role and behavioural constraints), **MD** (the same content in Markdown with structured headers), and **fewshot** (**basic** extended with worked task–call examples). For Plan-and-Act, planner and validator each have **basic** and **structured** variants; the worker uses a fixed prompt. Full prompt texts are provided in Appendix F.

Tool description format. Comprehensive descriptions specify each tool's purpose, parameter semantics, constraints, and failure conditions. **Minimal** descriptions reduce each tool to a single sentence, saving approximately 2,000 tokens across the 14 tools.

History scope. Under **run** scope, the full interaction history accumulates across all tasks in a session. Under **task** scope, each task starts with a fresh context.

3.4 Benchmark

We construct an expert-informed benchmark comprising six core task suites—Easy, Medium, Hard, History, Errors, and Cross—and two multi-server suites (Easy-Noise, Hard-Noise). Professional users of the design tool identified recurring hardware design operations, reviewed the task categories, and validated expected call sequences by executing them against the server. Task prompts reflect routine component-development operations. The full benchmark cannot be released because it encodes proprietary workflow details; representative examples are provided in Appendix B.

Independent tasks. (Appendix B.1–B.3) Easy, Medium, and Hard contain 40 self-contained tasks each and isolate the effect of task length: Easy requires 1 expected call, Medium requires 2, and Hard requires 3–5 calls with dependencies. Each prompt provides all required identifiers, such as the full VLNV of a component or the names of the ports being connected, so no information from prior turns is needed.

History. (Appendix B.4) History contains 40 tasks whose prompts omit entity identifiers and instead refer to prior tasks, for example, "Add the same port to the component created in the previous task." The agent must recover the relevant component and port information from earlier turns before acting.

Errors. (Appendix B.5) Errors contains 40 tasks. Twenty contain heavily misspelled prompts with the same intended content as Hard tasks, enabling a paired comparison of prompt noise. The remaining 20 describe requests for which the correct response is to make no tool call.

Cross. (Appendix B.7) Cross contains 40 tasks that combine dependency chains, cross-task references, pattern expansion, and error detection within a single multi-task session. For example, early tasks create components `sub1` and `sub2` with identical port sets; a later task asks to "create `sub3` like the others and wire it the same way"; and a final task requests an operation on a component that was never created, which the agent must reject rather than execute.

Noisy context. (Appendix B.8) Noisy context contains two 60-task suites evaluated on three models.

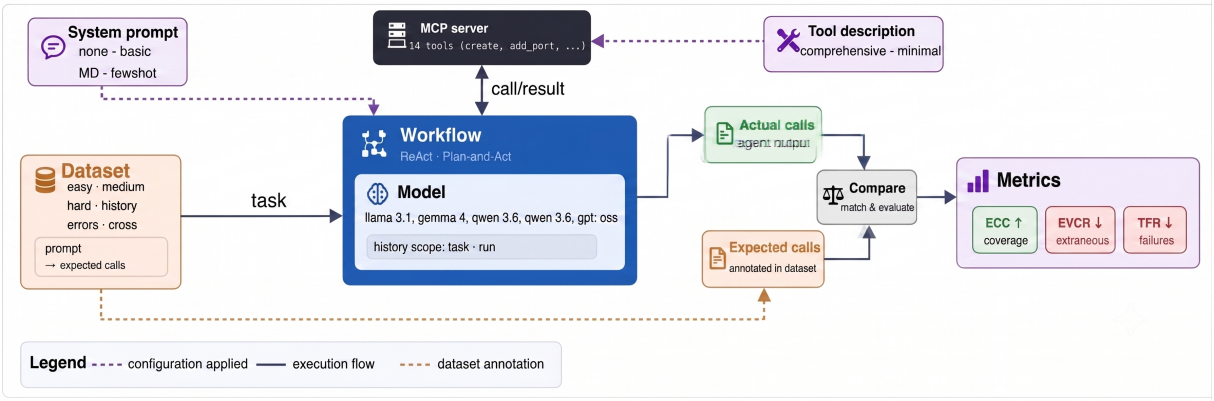

Figure 2: Evaluation pipeline. Each task prompt is submitted to the agent, which issues tool calls to the MCP server and receives results. The produced call sequence is compared against annotated expected calls to compute the metrics of Section 3.5.

Each suite interleaves one external-server task after every two system design tasks while exposing the design server alongside GDB debugging[1] and Git version-control[2] MCP servers. **Easy-Noise** uses the 40 Easy tasks; **Hard-Noise** uses the 40 Hard tasks. Because the system design tasks are unchanged, the clean suites provide a paired baseline for measuring routing errors and long-session effects.

3.5 Evaluation Metrics

Evaluating stateful agents requires call-level metrics rather than textual output assessment (Yao et al., 2024; Gao et al., 2025). We define four complementary metrics, interpreted jointly.

Expected Call Coverage (ECC, ↑) measures how much of the annotated work was completed:

$$\mathrm{ECC} = \frac{|\text{correctly executed expected calls}|}{|\text{expected calls}|}. \quad (1)$$

A call matches an expected call if it succeeds with the expected tool name and all expected argument values; each expected call is consumed at most once. Matching is order-insensitive because dependency-order violations are enforced by the server, which rejects invalid calls; such failures are accounted for in TFR. Tasks with an empty expected call set are excluded from ECC aggregation.

Extraneous Valid Call Ratio (EVCR, ↓) measures over-generation, i.e., successful calls that were not requested:

$$\mathrm{EVCR} = \frac{|\text{successful non-expected calls}|}{|\text{successful calls}|} \quad (2)$$

If an agent issues no successful calls, EVCR is defined as 0 by convention.

Tool Failure Rate (TFR, ↓) measures the fraction of calls rejected by the server:

$$\mathrm{TFR} = \frac{|\text{failed calls}|}{|\text{total calls}|} \quad (3)$$

If an agent issues no calls, TFR is defined as 0.

No-Call Accuracy (NCA, ↑) measures correct abstention on tasks with an empty expected call set:

$$\mathrm{NCA} = \frac{|\text{empty-expected tasks with no tool call}|}{|\text{empty-expected tasks}|} \quad (4)$$

Evaluation protocol. Tasks are presented sequentially. After each task, we remove all mutations produced by the agent and restore the server to the canonical state for the next task by replaying the expected sequence of calls up to that task. This prevents one agent error from making later tasks impossible while preserving the intended state dependencies in History and Cross. Figure 2 illustrates the pipeline. For ReAct, all configuration combinations are evaluated on the six core suites (Easy through Cross); the noisy-context suites are evaluated on three representative models.

4 Results

4.1 Can Hardware Design Be Automated?

Table 2 reports, for each model, the ReAct configuration that maximises average ECC across the six core task suites, together with the resulting EVCR and TFR. These numbers should be interpreted as best-observed performance under configuration search, not as out-of-the-box model performance. Plan-and-Act is considered separately below.

[1] https://github.com/signal-slot/mcp-gdb

[2] https://github.com/modelcontextprotocol/servers/tree/main/src/git

Figure 3: Best-configuration ECC per model and task suite under ReAct.

	Config	ECC ↑	EVCR ↓	TFR ↓
Gemma 4 31B	R/MD/C	0.990	0.015	0.011
Gemma 4 26B	R/MD/C	0.958	0.021	0.061
GPT-OSS 20B	R/none/C	0.951	0.073	0.049
Qwen 3.5 27B	R/MD/M	0.880	0.036	0.048
Qwen 3.6 27B	R/MD/M	0.858	0.086	0.065
Gemma 4 E4B	R/MD/C	0.811	0.023	0.058
Llama 3.1 8B	T/fs/C	0.554	0.064	0.353

Table 2: Best single ReAct configuration per model, averaged over the six core task suites and ordered by ECC. Config notation: history / prompt / tools, where R = run, T = task, C = comprehensive, M = minimal, fs = few-shot.

The aggregate hides a strong interaction between model and task structure. Figure 3 breaks down ECC by task suite: the model gap is small on simple tasks and widens as tasks require more state recovery and dependency management. Gemma 4 E4B stays within 6–14% of Gemma 4 31B on five of six task suites, but on Cross the gap widens to 47% (0.517 vs. 0.972). The same pattern appears within a single model: Llama 3.1 8B reaches 0.753 average ECC on the independent task suites (Easy, Medium, Hard), but only 0.139 on the two suites that require carrying state across tasks (History, Cross), an 82% drop.

Best configuration is itself workload-dependent. Table 2 reports the configuration that maximises average ECC across all six core suites, but for Llama 3.1 8B this global optimum gives only 0.113 ECC on History, whereas the History-specific optimum reaches 0.538. Even the globally best configuration is not reached automatically: under a poor configuration, Gemma 4 E4B drops from 0.811 to 0.168 ECC on the same tasks. Thus, model choice sets the broad performance range, but configuration determines whether a model reaches that range in practice.

Figure 3 omits two important failure modes. Table 3 reports both. First, the Errors suite includes 20 tasks with no correct tool call; for these tasks ECC is undefined, so we report NCA, EVCR, and TFR. Gemma 4 31B is nearly perfect, but other models either call unnecessary valid tools, issue failing calls, or both. Second, co-locating the design server with GDB and Git servers leaves ECC largely unchanged at matched configuration for the three models tested, but raises EVCR for all three, suggesting that a larger tool context mainly increases extraneous routing errors rather than reducing coverage.

	No-call (Errors)			Noise (Hard)	
	NCA	EVCR	TFR	clean	noisy
Gemma 4 31B	0.80	0.00	0.05	—	—
Gemma 4 26B	0.40	0.05	0.55	0.00	0.16
GPT-OSS 20B	0.25	0.58	0.13	—	—
Qwen 3.5 27B	0.70	0.15	0.17	—	—
Qwen 3.6 27B	0.45	0.20	0.37	—	—
Gemma 4 E4B	0.65	0.10	0.25	0.00	0.05
Llama 3.1 8B	0.00	0.25	0.82	0.12	0.22

Table 3: NCA, EVCR, and TFR on the 20 no-call tasks in Errors, at each model's best configuration. Right: EVCR on Hard vs. Hard-Noise under task scope and matched configuration for the three models tested. ECC changes are small and omitted for space.

Model and task structure are the two primary factors. The remaining results ask which agent-configuration choices still matter once those two are fixed.

4.2 How Much Does Agent Configuration Matter?

System prompt. Markdown-formatted instructions (MD) are the best prompt for five of the seven models, although the margin over none/basic is generally small (under 0.04 ECC). The main exception is few-shot prompting. For five models, few-shot changes ECC by at most 0.04 on average and often reduces EVCR and TFR; for Gemma 4 31B and Gemma 4 E4B, however, it causes severe inaction. Gemma 4 31B drops from 0.956 ECC with the best non-few-shot prompt to 0.571 with few-shot, and Gemma 4 E4B drops from 0.731 to 0.179. In both cases EVCR and TFR fall together with ECC, indicating that the models stop acting rather than acting incorrectly. In this benchmark, prompt engineering has an asymmetric risk profile: gains over simpler prompts are modest for most models, while a poorly matched few-shot prompt can be catas-

trophic. Detailed prompt ablations are reported in Appendix D.1.

Tool descriptions. Comprehensive tool descriptions provide the most consistent configuration benefit. Holding all other choices fixed, switching to minimal descriptions raises TFR for every model, roughly doubling it for most. The effect on ECC is less uniform, but the reliability gain suggests that parameter semantics, constraints, and failure conditions help models construct valid calls. Across the 42 model–task-suite combinations in Figure 3, the configuration that maximises ECC on that suite uses comprehensive descriptions in 35 cases (83%). Each model's globally best configuration also uses comprehensive descriptions in five of seven cases. Full tool-description ablations are reported in Appendix D.2.

Context and history scope. On independent tasks, history scope has little effect for six of the seven models: ECC shifts by at most 0.031. Llama 3.1 8B is the exception. On Easy, Medium, and Hard, cumulative history cuts its ECC from 0.667 to 0.192, a 71% drop, while TFR also falls from 0.194 to 0.070. This pattern indicates silence rather than more frequent invalid calls. For constrained models, retaining irrelevant history can therefore be actively harmful. Full history-scope results are reported in Appendix D.3.

Architecture. Table 4 compares ReAct with Plan-and-Act using Gemma 4 26B as planner. This is a pipeline-level comparison: Plan-and-Act can improve weak workers partly by delegating decomposition to a stronger model. With Llama 3.1 8B as worker, decomposition raises average ECC from 0.554 to 0.718, with the largest gains on History and Cross. With Gemma 4 26B as worker on the same suites, decomposition does not help and reduces coverage, suggesting that strong single-agent workers benefit from retaining full session context. On longer Hard-Noise sessions under cumulative history, however, Plan-and-Act recovers coverage from 0.858 to 0.950 at the cost of higher EVCR. Thus, multi-agent decomposition is most useful when the worker is weak or the session is long enough for single-agent context accumulation to become a bottleneck.

Worker	Suite	ReAct	Plan-and-Act
Llama 3.1 8B	Avg. (6 core)	0.554	**0.718**
Llama 3.1 8B	History	0.113	**0.650**
Llama 3.1 8B	Cross	0.166	**0.450**
Gemma 4 26B	Hard-Noise (60)	0.858	**0.950**

Table 4: ECC under ReAct vs. Plan-and-Act, at each worker's best configuration (averaged over the six core suites). The Hard-Noise row uses a fixed run-scope configuration, isolating the effect of session length.

4.3 Stability

All main experiments use each model's default temperature (1.0, except 0.8 for Llama 3.1 8B). Re-running all configurations on three representative datasets across temperatures and repeating a subset of configurations under identical conditions confirms that the main effects are not artefacts of sampling. Across temperatures 0, 0.5, and 1.0, model rankings are preserved and ECC shifts remain small; for Gemma 4 26B, ECC ranges from 0.881 to 0.896. Run-to-run variance is low for Gemma 4 26B and moderate for weaker models; the largest outlier corresponds to the few-shot collapse described above. Full stability results are reported in Appendix C.

5 Conclusion

We evaluated whether LLM agents can automate expert-defined hardware design tasks through MCP tool calling. On our benchmark, the strongest open-source models achieve near-perfect expected-call coverage under their best configuration, including on multi-step sessions with implicit state and co-located GDB and Git servers.

Model choice primarily determines the achievable performance range, while system configuration determines whether that performance is actually reached in practice.

Our results suggest five deployment practices: test prompts on the target model; use comprehensive tool descriptions; benchmark models on workload-representative tasks; manage context for constrained models; and reserve multi-agent decomposition for weak workers or long sessions.

These findings indicate that local LLM agents are practical for structured, stateful hardware component-editing workflows when the tool-calling pipeline is appropriately configured.

6 Limitations

The main limitation is the scope of our context-management exploration. We tested only the binary choice between per-task and cumulative history; mechanisms such as a variable memory window—summarising or discarding older turns rather than retaining or dropping all of them—could potentially improve weaker models on session-dependent tasks, and our results hint at this but do not test it. Relatedly, our session-dependent datasets are limited to tens of tasks; longer sessions might reveal additional context-management effects for models beyond Llama 3.1 8B, which was the only model showing sensitivity to history scope at this session length.

Second, EVCR counts extraneous successful calls but does not weight their severity: an unnecessary user parameter and an incorrect subcomponent connection contribute equally to the ratio, though the latter has far greater impact on design correctness. A severity-weighted variant is left for future work.

Third, we do not systematically measure response latency. Local inference time scales with model size, and a latency-sensitive deployment might reasonably prefer a smaller model at some cost in coverage—a trade-off our metrics do not capture.

Finally, the MCP server mirrors the data model and dependency rules of the proprietary design tool but does not call its production APIs; the results therefore characterise agent behaviour against a faithful replica, and some gap with production behaviour should be expected. Neither the server implementation nor the benchmark tasks can be released, as both encode proprietary component specifications and workflow details; Appendix B provides representative task examples to support methodological reproduction.

Acknowledgments

We thank Huawei for the opportunity to carry out this work during Leonardo's internship. We are especially grateful to Rahul Setia, Leonardo's supervisor at Huawei, for his guidance throughout the work, and to Johan Hokfelt for helping make this submission possible.

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, and 1 others. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*.

Anthropic. 2024a. Introducing the model context protocol.

Anthropic. 2024b. Tool use with claude: Best practices for tool definitions.

Tara Bogavelli, Roshnee Sharma, and Hari Subramani. 2025. Agentarch: a comprehensive benchmark to evaluate agent architectures in enterprise. *arXiv preprint arXiv:2509.10769*.

Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. Chipgpt: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019*.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, and 1 others. 2024. Workarena: How capable are web agents at solving common knowledge work tasks? *arXiv preprint arXiv:2403.07718*.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2025. Plan-and-act: Improving planning of agents for long-horizon tasks. *arXiv preprint arXiv:2503.09572*.

Yonggan Fu, Yongan Zhang, Zhongzhi Yu, Sixu Li, Zhifan Ye, Chaojian Li, Cheng Wan, and Yingyan Celine Lin. 2023. Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9. IEEE.

Xuanqi Gao, Siyi Xie, Juan Zhai, Shiqing Ma, and Chao Shen. 2025. Mcp-radar: A multi-dimensional benchmark for evaluating tool use capabilities in large language models. *arXiv preprint arXiv:2505.16700*.

Google DeepMind. 2026. Gemma 4: Open models for everyone.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Jia He, Mukund Rungta, David Koleczek, Arshdeep Sekhon, Franklin X Wang, and Sadid Hasan. 2024. Does prompt formatting have any impact on llm performance? *arXiv preprint arXiv:2411.10541*.

Kung-Hsiang Huang, Akshara Prabhakar, Sidharth Dhawan, Yixin Mao, Huan Wang, Silvio Savarese, Caiming Xiong, Philippe Laban, and Chien-Sheng Wu. 2025. Crmarena: Understanding the capacity of llm agents to perform professional crm tasks in realistic environments. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3830–3850.

Antti Kamppi and 1 others. 2012. Kactus2: A graphical ip-xact tool for design and configuration of embedded processing systems. In *SAMOS*.

Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same task, more tokens: the impact of input length on the reasoning performance of large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15339–15353.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, and 1 others. 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.

Ollama. 2024. Ollama: Run large language models locally.

Qwen Team, Alibaba Cloud. 2026a. Qwen3.5: Towards native multimodal agents.

Qwen Team, Alibaba Cloud. 2026b. Qwen3.6-plus: Towards real world agents.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076.

Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and 1 others. 2025. Mcp-bench: Benchmarking tool-using llm agents with complex real-world tasks via mcp servers. *arXiv preprint arXiv:2508.20453*.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Lechen Zhang, Tolga Ergen, Lajanugen Logeswaran, Moontae Lee, and David Jurgens. 2024. Sprig: Improving large language model performance by system prompt optimization. *arXiv preprint arXiv:2410.14826*.

A Agent Architecture Diagrams

A.1 ReAct

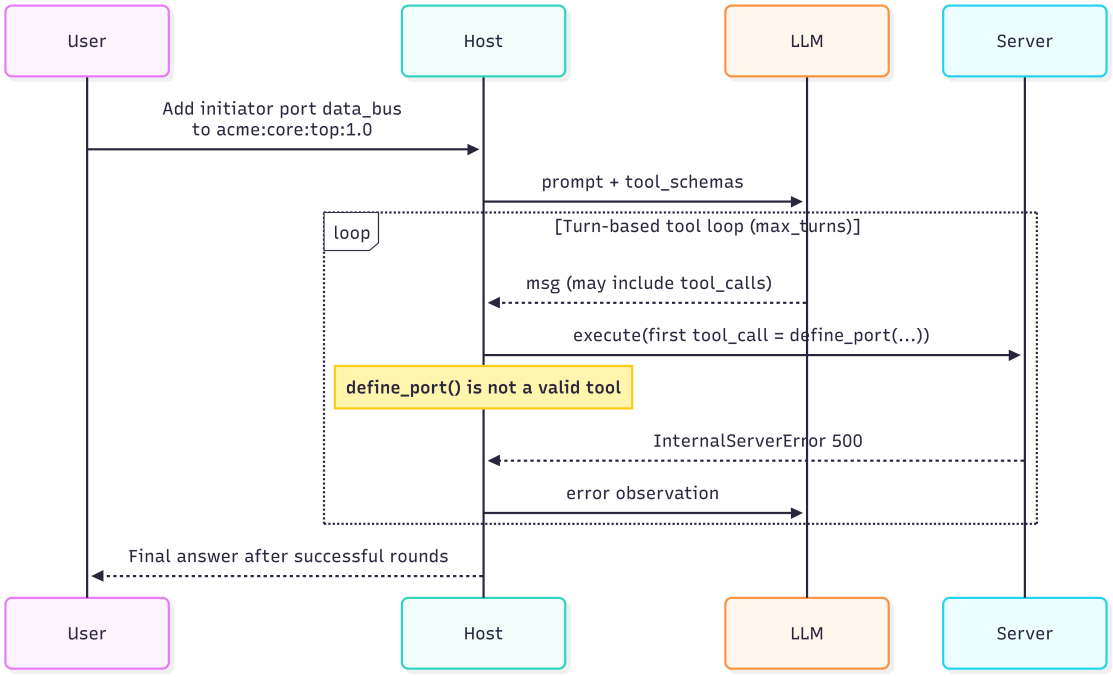

Figure 4: ReAct sequence diagram. The host submits the task prompt and tool schemas to the LLM. Within a turn-bounded loop, the LLM returns a response that may include a tool call; the host executes the first call, appends the result, and repeats. The loop terminates when no further tool call is produced or the turn limit is reached.

A.2 Plan-and-Act

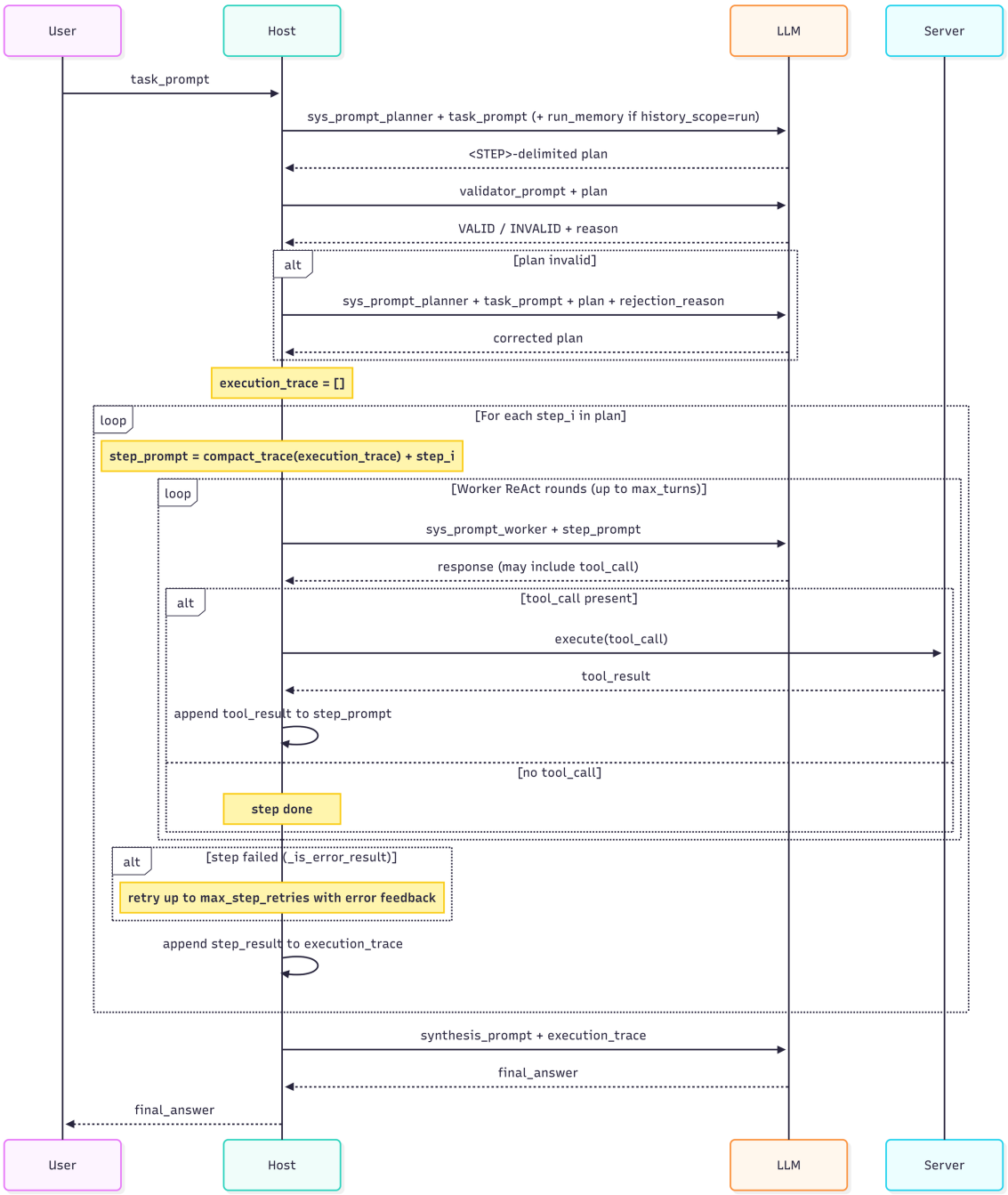

Figure 5: Plan-and-Act sequence diagram. The planner produces a <STEP>-delimited plan; the validator checks it with one replan permitted. Each step is executed by an independent ReAct worker with a compact trace of prior results. A synthesis call assembles the final answer.

B Task Examples

B.1 Easy

EASY_001

"Create a component with vendor acme, library core, name top, and version 1.0."

```
create_component(acme, core, top, 1.0)
```

EASY_002

"Add an input signal port named clk of width 1 to component acme:core:top:1.0."

```
[on acme:core:top:1.0]
add_signal_port(clk, in, w=1)
```

EASY_003

"Add an initiator transactional port named tx to component acme:core:top:1.0."

```
[on acme:core:top:1.0]
add_transactional_port(tx, initiator)
```

EASY_004

"Add an output signal port named data_out of width 8 to component acme:core:top:1.0."

```
[on acme:core:top:1.0]
add_signal_port(data_out, out, w=8)
```

EASY_005

"Add a user parameter named DATA_WIDTH of type integer with default value 32 to component acme:core:top:1.0."

```
[on acme:core:top:1.0]
add_user_parameter(DATA_WIDTH, integer,
=32)
```

B.2 Medium

MED_001

"Create a component riscv_core (vendor openhw, lib cpu, v 1.0) and add a transactional initiator port named data_bus."

```
create_component(openhw, cpu,
riscv_core, 1.0)
add_transactional_port(data_bus,
initiator)
```

MED_002

"The riscv_core also needs a clock input. Add a 1-bit input signal port named clk, then save."

```
[on openhw:cpu:riscv_core:1.0]
add_signal_port(clk, in, w=1)
save_component
```

MED_003

"Create a component sram (vendor openhw, lib mem, v 1.0) and add a transactional target port slave_port with a memory implementation called mem of 1024 bytes."

```
create_component(openhw, mem, sram,
1.0)
add_transactional_port(slave_port,
target, impl=memory, mem=mem,
bytes=1024)
```

MED_004

"Complete the sram component by adding a 1-bit input signal named clk, then save it."

```
[on openhw:mem:sram:1.0]
add_signal_port(clk, in, w=1)
save_component
```

B.3 Hard

HARD_001

"Let's start our SoC design by creating the main CPU. Please define a component named riscv_tile from vendor acme, library ip_cores, version 2.1.0. Add an initiator transactional port called m_axi for memory access and a signal input port for the clk with a width of 1 and multiplicity of 1. Finally, save the component to disk."

```
create_component(acme, ip_cores,
riscv_tile, 2.1.0)
add_transactional_port(m_axi,
initiator)
add_signal_port(clk, in, w=1)
save_component
```

HARD_002

"We need a memory controller now. Create sram_ctrl (vendor: acme, lib: mem, v: 1.0.0). It needs a target transactional port named s_axi using a memory implementation called mem of 30 bytes. Also, add a 1-bit input signal port for rst_n and a 1-bit input clk. Save it when done."

```
create_component(acme, mem, sram_ctrl,
1.0.0)
add_transactional_port(s_axi, target,
impl=memory, mem=mem, bytes=30)
add_signal_port(rst_n, in, w=1)
add_signal_port(clk, in, w=1)
save_component
```

HARD_003

"Let's define a system interconnect component named axi_interconnect (vendor: acme, lib: bus, v: 1.1). Add one initiator transactional port m0_port and two target transactional ports s0_port and s1_port with user_defined implementation. Then save the component."

```
create_component(acme, bus,
axi_interconnect, 1.1)
add_transactional_port(m0_port,
```

```
initiator)
add_transactional_port(s0_port, target,
impl=user_defined)
add_transactional_port(s1_port, target,
impl=user_defined)
save_component
```

B.4 History

Tasks run in session order; later tasks omit identifiers and must recover them from earlier turns.

HIST_001

"Create a RISC-V core openhw:cpu:riscv_v5:1.0 and add a transactional initiator port data_master."

```
create_component(openhw, cpu, riscv_v5,
1.0)
add_transactional_port(data_master,
initiator)
```

HIST_002

"Add another transactional initiator port called instr_master."

Note: no id given; recover openhw:cpu:riscv_v5:1.0 from HIST_001.

[on openhw:cpu:riscv_v5:1.0]

```
add_transactional_port(instr_master,
initiator)
```

HIST_003

"Create system_sram (generic, mem, 2.1) with a target port mem_port, memory mem, 65536 bytes."

```
create_component(generic, mem,
system_sram, 2.1)
add_transactional_port(mem_port,
target, impl=memory, mem=mem,
bytes=65536)
```

HIST_004

"Create axi_interconnect (generic, bus, 1.0) and add a user_defined target port slave_0."

```
create_component(generic, bus,
axi_interconnect, 1.0)
add_transactional_port(slave_0, target,
impl=user_defined)
```

HIST_005

"Go back to the RISC-V core. Add a 1-bit input clock and a 1-bit input reset_n."

Note: "the RISC-V core" = openhw:cpu:riscv_v5:1.0 from HIST_001.

[on openhw:cpu:riscv_v5:1.0]

```
add_signal_port(clock, in, w=1)
add_signal_port(reset_n, in, w=1)
```

B.5 Errors — misspelled vs. clean

Each misspelled Errors prompt mirrors a Hard prompt with heavy orthographic noise but identical intent. The pairs below share the same expected calls, isolating the effect of prompt noise.

HARD_001 vs ERR_001

Clean (Hard):

"*Let's start our SoC design by creating the main CPU. Please define a component named riscv_tile from vendor acme, library ip_cores, version 2.1.0. Add an initiator transactional port called m_axi and a signal input port for clk with width 1 and multiplicity 1. Finally, save the component to disk.*"

Misspelled (Errors):

"*Let's start our SoC design by cretaing the main CPU. Please defnie a compnnt named riscv_tile from vendor acme, library ip_cores, version 2.1.0. Add an initator transactonal prt called m_axi and a siganl inpt prt for clk with width 1 and multiplicty 1. Finally, saev the compnnt to disk.*"

Identical expected calls:

```
create_component(acme, ip_cores,
riscv_tile, 2.1.0)
add_transactional_port(m_axi,
initiator)
add_signal_port(clk, in, w=1)
save_component
```

HARD_002 vs ERR_002

Clean (Hard):

"*We need a memory controller now. Create sram_ctrl (acme, mem, 1.0.0). It needs a target transactional port named s_axi using a memory implementation called mem of 30 bytes. Also add a 1-bit input signal port for rst_n and a 1-bit input clk. Save it when done.*"

Misspelled (Errors):

"*We need a memroy controller now. Cretae sram_ctrl (acme, mem, 1.0.0). It needs a trgt transactinal prt named s_axi usng a memroy implmentation called mem of 30 bytes. Also add a 1-bit input siganl prt for rst_n and a 1-bit input clk. Saev it when done.*"

Identical expected calls:

```
create_component(acme, mem, sram_ctrl,
1.0.0)
add_transactional_port(s_axi, target,
impl=memory, mem=mem, bytes=30)
add_signal_port(rst_n, in, w=1)
add_signal_port(clk, in, w=1)
save_component
```

HARD_003 vs ERR_003

Clean (Hard):

"Let's define a system interconnect component named axi_interconnect (acme, bus, 1.1). Add one initiator transactional port m0_port and two target transactional ports s0_port and s1_port with user_defined implementation. Then save the component."

Misspelled (Errors):

"Let's defnie a system interconect compnent axi_interconnect (acme, bus, 1.1). Add one initator transactonal prt m0_port and two trgt transactonal prts s0_port and s1_port with user_defned implmentation. Then saev the compnent."

Identical expected calls:

```
create_component(acme, bus,
axi_interconnect, 1.1)
add_transactional_port(m0_port,
initiator)
add_transactional_port(s0_port, target,
impl=user_defined)
add_transactional_port(s1_port, target,
impl=user_defined)
save_component
```

B.6 Errors — no-call

The correct behaviour is to issue no call and explain why.

ERR_004

"Create a new component for our DMA block with vendor acme, library interconnect, name dma."

no call — `version is required but missing; cannot be inferred.`

ERR_005

"Add a signal port called irq to component acme:cpu:core:1.0."

no call — `direction and width missing; component also absent from state.`

ERR_006

"On component acme:mem:sram_ctrl:2.1, add a transactional port named cfg as a target backed by memory."

no call — `memory implementation needs a memory name and size; neither given.`

ERR_007

"Save the component."

no call — `no component identified.`

ERR_008

"Add a subcomponent named cpu0 into acme:accel:fft:3.0."

no call — `child_component_id (what to instantiate) is missing.`

B.7 Cross

CROSS_001

"Create four CPU shells (vendor Comp, library egg, version 1.0, names cpu1..cpu4)."

```
create_component(Comp, egg, cpu1, 1.0)
create_component(Comp, egg, cpu2, 1.0)
create_component(Comp, egg, cpu3, 1.0)
create_component(Comp, egg, cpu4, 1.0)
```

CROSS_002

"To all components, add a single input signal port called p1."

Note: "all" = cpu1..cpu4 from CROSS_001 (one call each).

```
add_signal_port(p1, in, w=1)
add_signal_port(p1, in, w=1)
add_signal_port(p1, in, w=1)
add_signal_port(p1, in, w=1)
```

CROSS_003

"In cpu4 add transactional initiator ports ibus1..ibus4."

[on Comp:egg:cpu4:1.0]

```
add_transactional_port(ibus1,
initiator)
add_transactional_port(ibus2,
initiator)
add_transactional_port(ibus3,
initiator)
add_transactional_port(ibus4,
initiator)
```

CROSS_004

"To cpu1, add subcomponents: cpu2 as sub1, cpu3 as sub2, cpu4 as sub3."

[on Comp:egg:cpu1:1.0]

```
add_subcomponent(cpu2 as sub1)
add_subcomponent(cpu3 as sub2)
add_subcomponent(cpu4 as sub3)
```

CROSS_005

"Add to it also the same initiator ports of cpu4."

Note: "it"=cpu1; replicate cpu4's ibus1..ibus4 (from CROSS_003).

[on Comp:egg:cpu1:1.0]

```
add_transactional_port(ibus1,
initiator)
add_transactional_port(ibus2,
initiator)
add_transactional_port(ibus3,
initiator)
```

```
add_transactional_port(ibus4,
initiator)
```

CROSS_006

"Hook all ibus ports in sub3 to the same-named ports of the parent (cpu1)."

Note: sub3 is an instance of cpu4 (CROSS_004), which has ibus1..ibus4.

[on Comp:egg:cpu1:1.0]

```
connect(sub3.ibus1 ->
hierarchical.ibus1)
connect(sub3.ibus2 ->
hierarchical.ibus2)
connect(sub3.ibus3 ->
hierarchical.ibus3)
connect(sub3.ibus4 ->
hierarchical.ibus4)
```

B.8 Noisy context

Every two design tasks, one external-server (Git or GDB) task is inserted; the agent must route to the right server. Routing mistakes surface as EVCR, not ECC.

EASY_NOISE_IPXACT_001

"Create a component with vendor acme, library core, name top, version 1.0."

```
create_component(acme, core, top, 1.0)
```

EASY_NOISE_GIT_001

"Show the last 5 commits in the repository at /home/user/.../dummy_repo."

```
git_log(repo, max_count=5)
```

EASY_NOISE_IPXACT_002

"Add an input signal port named clk of width 1 to component acme:core:top:1.0."

[on acme:core:top:1.0]

```
add_signal_port(clk, in, w=1)
```

EASY_NOISE_GDB_001

"Start a new debugging session."

```
gdb_start()
```

HARD_NOISE_IPXACT_001

"(same as HARD_001: create riscv_tile, add m_axi and clk, save)."

```
create_component(acme, ip_cores,
riscv_tile, 2.1.0)
add_transactional_port(m_axi,
initiator)
add_signal_port(clk, in, w=1)
save_component
```

HARD_NOISE_GIT_001

"List all remote branches in the repository at /home/user/.../dummy_repo."

```
git_branch(repo, remote)
```

HARD_NOISE_GDB_001

"Show the source at the current execution point in debugging session 1."

```
gdb_list_source(session=1)
```

C Stability Experiments

C.1 Temperature Sweep

	$T = 0$			$T = 0.5$			T=**default**		
Model	ECC	EVCR	TFR	ECC	EVCR	TFR	ECC	EVCR	TFR
Gemma 4 26B	0.896	0.055	0.094	0.889	0.055	0.111	**0.881**	0.056	**0.105**
Gemma 4 E4B	0.539	0.044	0.129	0.523	0.042	0.127	**0.553**	0.046	0.144
Llama 3.1 8B	**0.247**	0.058	0.268	0.235	0.052	**0.264**	0.243	0.056	0.268

Table 5: ECC and TFR across temperatures 0, 0.5, and each model's default (1.0, except 0.8 for Llama 3.1 8B), averaged over the three datasets covered by the sweep (Cross, Errors, Hard) and over all 16 system-prompt $\times$ tool-description $\times$ history-scope configurations per model. Rankings and absolute performance are stable across temperatures for every model; lowering temperature does not consistently reduce TFR.

C.2 Run-to-Run Variance

Model	$\bar{\sigma}_{\mathrm{ECC}}$	$\sigma_{\max,\mathrm{ECC}}$
Gemma 4 26B	**0.025**	0.076
Llama 3.1 8B	0.051	0.219
Gemma 4 E4B	0.059	**0.494**

Table 6: Mean and maximum standard deviation of ECC across two repeated runs of every system-prompt $\times$ tool-description $\times$ history-scope configuration (16 per model), on Cross, Errors, and Hard — the three most demanding suites. The Gemma 4 E4B outlier ($\sigma_{\max} = 0.494$) is the `run`-scope, few-shot, minimal-tools configuration, the same one responsible for the few-shot collapse reported in Section 4.2, confirming that effect is genuine rather than a sampling artefact.

All configurations, hardest suites (2 runs each).

Model	**Config**	ECC	EVCR	TFR
Gemma 4 26B	best	**0.973**	0.027	**0.013**
Gemma 4 26B	worst	0.829	**0.017**	0.067
Llama 3.1 8B	best	**0.495**	0.106	0.291
Llama 3.1 8B	worst	0.034	**0.001**	**0.028**

Table 7: Mean ECC, EVCR, and TFR across 10 repeated runs of each model's best and worst configuration, averaged over all six core suites (10 tasks each). Standard deviations of ECC across runs: 0.008 (Gemma best), 0.038 (Gemma worst), 0.033 (Llama best), 0.109 (Llama worst) — the latter driven by near-total collapse (9 of 10 runs at ECC ≈ 0) rather than ordinary run-to-run noise.

Best/worst configuration, repeated subset (10 runs each).

D Additional Configuration Results

D.1 Prompt Ablations

	none			basic			MD			few-shot		
Model	ECC	EVCR	TFR	ECC	EVCR	TFR	ECC	EVCR	TFR	ECC	EVCR	TFR
Gemma 4 31B	0.935	0.039	0.076	0.913	0.010	0.056	**0.956**	0.021	0.061	0.571	0.006	0.032
Gemma 4 26B	0.907	0.044	0.086	0.910	0.020	0.063	**0.938**	0.027	0.063	0.897	0.034	0.071
GPT-OSS 20B	**0.861**	0.078	0.104	0.784	0.030	0.068	0.859	0.042	0.061	0.741	0.030	0.072
Qwen 3.5 27B	0.847	0.063	0.068	0.827	0.031	0.058	**0.852**	0.034	0.057	0.825	0.032	0.054
Qwen 3.6 27B	0.839	0.063	0.106	0.798	0.030	0.063	**0.853**	0.058	0.060	0.801	0.036	0.056
Gemma 4 E4B	0.675	0.036	0.148	0.694	0.031	0.124	**0.731**	0.026	0.135	0.179	0.016	0.072
Llama 3.1 8B	0.293	0.031	0.301	0.284	0.049	0.255	0.290	0.056	0.229	**0.339**	0.034	0.211

Table 8: System-prompt ablation, averaged over the six core suites, tool-description formats, and history scopes. The few-shot collapse for Gemma 4 31B ($0.956 \rightarrow 0.571$) and Gemma 4 E4B ($0.731 \rightarrow 0.179$) is visible as a coverage drop with EVCR and TFR falling together (inaction, not error).

D.2 Tool-Description Ablations

	comprehensive			minimal		
Model	ECC	EVCR	TFR	ECC	EVCR	TFR
Gemma 4 31B	0.819	0.019	0.031	**0.868**	0.020	0.081
Gemma 4 26B	**0.932**	0.028	0.048	0.894	0.034	0.093
GPT-OSS 20B	**0.859**	0.046	0.057	0.764	0.044	0.096
Qwen 3.5 27B	**0.845**	0.034	0.040	0.830	0.046	0.078
Qwen 3.6 27B	**0.823**	0.037	0.046	0.822	0.057	0.097
Gemma 4 E4B	**0.610**	0.028	0.073	0.530	0.026	0.167
Llama 3.1 8B	**0.312**	0.050	0.225	0.290	0.035	0.273

Table 9: Tool-description ablation, averaged over the six core suites, system prompts, and history scopes. Minimal descriptions raise TFR for every model (roughly doubling it in most cases); the ECC effect is less uniform.

D.3 History-Scope Ablations

	run			task		
Model	ECC	EVCR	TFR	ECC	EVCR	TFR
Gemma 4 31B	**0.858**	0.014	0.037	0.829	0.024	0.075
Gemma 4 26B	**0.918**	0.029	0.071	0.909	0.034	0.071
GPT-OSS 20B	**0.862**	0.051	0.061	0.760	0.039	0.091
Qwen 3.5 27B	**0.864**	0.047	0.038	0.811	0.033	0.081
Qwen 3.6 27B	**0.823**	0.051	0.060	0.822	0.042	0.083
Gemma 4 E4B	**0.621**	0.031	0.125	0.519	0.023	0.115
Llama 3.1 8B	0.157	0.015	0.084	**0.445**	0.070	0.413

Table 10: History-scope ablation, averaged over the six core suites, system prompts, and tool-description formats. Llama 3.1 8B is the only model that prefers per-task scope; all others tolerate or benefit from cumulative history at this session length.

E Tool Description Example

The following shows the `add_signal_port` tool in both description formats.

`add_signal_port --- comprehensive`

Add a signal port to an existing component.
The component must already exist and the port name must be unique. Direction must be 'in', 'out', or 'inout'. Width and multiplicity must be greater than zero. Default multiplicity value is 1.
On success, the new signal port becomes available for future connections.

`add_signal_port --- minimal`

Add a signal port to a component.

F System Prompts

All prompts are reproduced verbatim. Section F.1 contains the ReAct system prompts (`none`, `basic`, `MD`, `fewshot`). Section F.2 contains the Plan-and-Act stage prompts: planner (`basic`, `structured`), validator (`basic`, `structured`), and worker (fixed).

F.1 ReAct Prompts

`none`

No system prompt. The model receives only the task instruction and the tool schemas.

`basic`

You are an autonomous system-design agent operating in a tool-enabled environment. The server exposes tools that create, inspect, connect, delete, load, and persist components and their internal elements.

If the user requests creation, modification, connection, deletion, loading, or persistence of components or elements, you MUST call the appropriate tool when all required arguments are available.

If the user asks to rename or edit an element and there is no direct update tool for that operation, treat the request as deleting the existing element and recreating the same element with the requested change, while keeping all other retrievable fields unchanged. For example, changing the multiplicity of a signal port means deleting that port and recreating it with the new multiplicity and the same remaining attributes.

If the request appears incomplete, then follow this rule: if you are confident that the request contains a conceptual error, do NOT call any tool and explain the problem; if you are unsure, first use inspection tools (e.g., list_components, get_component_details, or get_element) and then decide how to proceed.

If some required tool arguments are missing after relevant inspection, DO NOT invent values and DO NOT call the tool. Instead, inform the user which required fields are missing.

You MUST NOT respond in natural language when a tool can be correctly used.

`MD`

Role
You are an autonomous design agent operating in a tool-enabled environment. The server exposes tools that create, inspect, connect, delete, load, and persist components and their elements.

Behavior
- Call one tool per turn.
- Do not invent argument values. Use only values stated in the task or retrieved from tools.
- If required arguments are missing after inspection, do not call the tool.
- Complete all operations stated in the task — do not stop after the first tool call.
- If the request is invalid or impossible, say so clearly without calling any tool.

Hierarchical connections
Use ”hierarchical” as the instance name only when the endpoint is a top-level port of the parent component itself, not a port of a subcomponent.

Invalid tool calls
If a tool call fails, do not retry with the same arguments. Read the error message and adjust before trying again.

`fewshot`

You are an autonomous system-design agent operating in a tool-enabled environment. The server exposes tools that create, inspect, connect, delete, load, and persist components and their elements. If the user requests creation, modification, connection, deletion, loading, or persistence of components or elements, you MUST call the appropriate tool when all required arguments are available. If required arguments are missing after inspection, do NOT invent values — inform the user instead. You MUST NOT respond in natural language when a tool can be correctly used.

—

EXAMPLE 1 — Apply the same operation to every component listed. Do not stop early.

User: Add a 1-bit input signal port named clk to each of these components: nordic:rf:tx_core:1.0, nordic:rf:rx_core:1.0, nordic:rf:pll:1.0.

[calls add_signal_port: component_id=nordic:rf:tx_core:1.0,port_name=clk, direction=in,width=1,multiplicity=1]
Tool: {”ok”: true}
[calls add_signal_port: component_id=nordic:rf:rx_core:1.0,port_name=clk, direction=in,width=1,multiplicity=1]
Tool: {”ok”: true}
[calls add_signal_port: component_id=nordic:rf:pll:1.0, port_name=clk,direction=in,width=1,multiplicity=1]
Tool: {”ok”: true}
A: Done. Port clk added to all three components.

—

EXAMPLE 2 — When a task has multiple distinct operations, complete all of them.

User: Inside arm:soc:top:2.0, connect cpu_inst’s axi_out port to bus_inst’s axi_in. Also connect the top-level sys_rst signal to cpu_inst’s rst_n. Then save.

[calls add_subcomponent_connection: parent_component_id=arm:soc:top:2.0, from_instance=cpu_inst,from_port=axi_out, to_instance=bus_inst,to_port=axi_in]
Tool: {”ok”: true}
[calls add_subcomponent_connection: parent_component_id=arm:soc:top:2.0, from_instance=hierarchical,from_port=sys_rst, to_instance=cpu_inst,to_port=rst_n]
Tool: {”ok”: true}
[calls save_component: component_id=arm:soc:top:2.0]
Tool: {”ok”: true}
A: Both connections established and component saved.

—

EXAMPLE 3 — The `"hierarchical"` instance name.

`"hierarchical"` refers exclusively to a port that belongs to the parent component itself, not to any subcomponent. Use it when wiring a top-level port down to a subcomponent, or up from a subcomponent to the top level. Never use it to connect two subcomponent instances to each other.

User: In risc:sys:chip_top:1.0, wire the top-level ext_irq input down to irq_ctrl_inst's irq_in port. Then wire irq_ctrl_inst's status_out up to the top-level debug_out port.

[calls add_subcomponent_connection: parent_component_id=risc:sys:chip_top:1.0, from_instance=hierarchical,from_port=ext_irq, to_instance=irq_ctrl_inst,to_port=irq_in]
Tool: {"ok": true}
[calls add_subcomponent_connection: parent_component_id=risc:sys:chip_top:1.0, from_instance=irq_ctrl_inst,from_port=status_out, to_instance=hierarchical,to_port=debug_out]
Tool: {"ok": true}
A: Both hierarchical connections established.

—

EXAMPLE 4 — If add_subcomponent_connection fails, do NOT modify child components or retry with guessed port names. Check get_component_details on the parent to verify what subcomponents and connections exist, then report what is missing. Do not invent ports or restructure components to force a connection.

User: In mips:platform:board:3.0, connect dma_inst's mem_req port to ram_inst's mem_ack port.

[calls add_subcomponent_connection: parent_component_id=mips:platform:board:3.0, from_instance=dma_inst,from_port=mem_req, to_instance=ram_inst,to_port=mem_ack]
Tool: {"ok": false, "error": {"code": "invalid_source_port", "message": "Port mem_req not found on instance dma_inst."}}
[calls get_component_details: component_id=mips:platform:board:3.0]
Tool: {"ok": true, "subcomponents": ["dma_inst", "ram_inst"], "connections": []}
A: The connection failed because port mem_req does not exist on dma_inst. I cannot create the connection. Please verify the correct port names on dma_inst and ram_inst.

—

Apply these patterns to every task. Always apply an operation to every component or element the task specifies — do not stop early. Always complete every distinct operation in the task before finishing. Always call save_component after structural changes when the task requests persistence. Use "hierarchical" only for ports that belong to the parent component itself. If a connection fails, inspect the parent with get_component_details and report what is missing — do not modify child components or invent port names.

F.2 Plan-and-Act Prompts

`Planner basic`

You are a planning agent responsible for decomposing a user request into a sequence of executable steps. Each step will be executed by a separate worker agent that has access to the following tools: {tools}

RULES:
- Each step must correspond to exactly ONE tool call.
- Steps must be ordered so that any value produced by step N is available to step N+1.
- If a step produces an identifier, name, or value that a later step needs, say so explicitly in that later step (e.g. 'using the component ID returned in the previous step').
- Preserve all identifiers, names, and parameters from the user request exactly as given.
- Do not invent missing values.
- Avoid pronouns like 'it' or 'that' — always repeat the full name or identifier.
- If the task requires only one tool call, output one step.

OUTPUT FORMAT:
- Write steps in natural language.
- Separate steps with the token <STEP> placed at the END of each step, including the last one.
- Do not number the steps.
- Do not add any preamble, explanation, or summary — output only the steps.

Example for a two-step task:
Create a component with id riscv_core, vendor openhw, library cpu, and version 1.0.<STEP>
Add a port named data_bus to the component with id riscv_core, using protocol transactional and port type initiator.<STEP>

`Planner structured`

You are a planning agent. Your job is to decompose a user request into a sequence of atomic steps, each executed by a separate worker agent. The worker has access to these tools — use them to understand what operations are possible, but write your steps in plain natural language describing the action, not the tool call: {tools}

STEP RULES:
- Each step must correspond to exactly ONE tool call.
- If a task needs 4 tool calls, write 4 steps. Never collapse multiple tool calls into one step.
- Order steps so that any value produced by step N is available to step N+1.
- When a later step needs an ID or value from an earlier step, say so explicitly (e.g. 'using the component ID returned in step 1').
- Copy all identifiers, names, and parameters from the user request exactly.
- Do not invent missing values.
- Never use pronouns like 'it' or 'that' — always repeat the full name.

RENAME AND EDIT OPERATIONS:
- There is no rename or update tool. Any request to rename or change a field of an existing element must be planned as: retrieve, delete, recreate.
- If the current field values are not given in the task, add a step to retrieve the element with get_element before the delete step.
- The recreate step must preserve all fields unchanged except the one being modified.
- Example: renaming signal port p2 to p_out on component X:
1. Retrieve port p2 from component X to get its current fields.<STEP>
2. Delete port p2 from component X.<STEP>
3. Add a signal port named p_out to component X with the same direction, width, and multiplicity as retrieved in step 1.<STEP>

INSPECT-BEFORE-CONNECT:
- If the instance names or port names needed for a connection step are not already known from the task description or prior steps, include a prior step that retrieves the component details first.
- Never guess or assume instance or port names.

THE "hierarchical" INSTANCE NAME — READ CAREFULLY:
- "hierarchical" is a reserved instance name meaning the

parent component itself, not any subcomponent instance.
- Use it ONLY when one endpoint of a connection is a port that belongs directly to the parent component.
- CORRECT: connecting subcomponent sub3 port ibus1 outward to the parent's own port ibus1:
from_instance="sub3",from_port="ibus1",
to_instance="hierarchical",to_port="ibus1"
- WRONG: connecting subcomponent bus_inst to subcomponent ram_inst:
WRONG: from_instance="hierarchical", from_port="m0_port",to_instance="ram_inst", to_port="s_axi"
RIGHT: from_instance="bus_inst",from_port="m0_port", to_instance="ram_inst",to_port="s_axi"

PURE-INSPECTION OR ERROR TASKS:
- If the task only requires reading or reporting state with no mutations, output zero steps.
- If the task references elements that do not exist, output zero steps.
- A plan with zero <STEP> tokens is valid.

OUTPUT FORMAT:
- Write steps in plain natural language.
- End every step, including the last, with the token <STEP>.
- Do not number steps.
- Do not add any preamble, heading, explanation, or summary.
- If the plan is empty, output nothing at all.

Validator basic

You are validating a plan for the following task: {task_prompt}
Available tools: {tool_names}
Plan to validate: {plan_text}

Check the plan against these criteria:
1. Every step maps to at least one available tool.
2. No step invents arguments not present in the task or derivable from prior steps.
3. Steps are in a logical order (no step depends on a result not yet produced).
4. No steps are duplicated or contradictory.

If the plan passes all criteria, respond with exactly: VALID
If it fails any criterion, respond with: INVALID: <brief reason>

Validator structured

You are validating a plan for the following task: {task_prompt}
Available operations (the worker uses these tools): {tool_names}
Plan to validate: {plan_text}

VALIDATION RULES:

1. TOOL COVERAGE — each step must describe an action that is achievable with at least one of the available tools. Steps are written in natural language: do not require them to contain the exact tool name.

2. ARGUMENT VALIDITY — no step may invent an argument value that is neither given in the task nor derivable from the result of a prior step. Cross-step references such as 'using the component ID returned in the previous step' are explicitly allowed and correct.

3. ORDERING — steps must be in a logical sequence where prerequisites come before the operations that depend on them. Do NOT reject a step merely because it references a value produced by an earlier step.

4. NO INVENTED REQUIREMENTS — only reject the plan for what it does wrong or omits relative to the task. Do not add requirements that are not in the task.

5. EMPTY PLAN — a plan with no steps is valid when the task requires no mutations. Do not reject an empty plan solely because it has no steps.

Respond with exactly one of:
VALID
INVALID: <one concise reason referencing the specific step or argument that fails>

Worker prompt (fixed, not an experimental axis)

You are an execution agent operating in a tool-enabled environment. You are executing exactly ONE step of a larger plan.

Your context contains:
- Results from previous steps (if any) — use those values directly, do not re-derive them.
- The current step you must execute.

SCOPE RULES:
- Execute ONLY what the current step explicitly describes. Nothing more.
- Do NOT inspect components, add ports, modify state, or call any tool that the step does not explicitly request.
- Do NOT attempt to fix or improve results from previous steps.
- If the step says to retrieve component details, call that tool only and report what it returned. Do not act further on the result.
- Call at most one tool per step unless the step explicitly lists two actions.

ARGUMENT RULES:
- Use only argument values stated in the step or present in prior step results.
- Do not invent, guess, or supply default values for missing arguments.
- If a required argument is missing, do NOT call the tool. Report which argument is missing.

RENAME AND EDIT OPERATIONS:
- There is no rename or update tool. If the step says to rename or edit an element, execute it as delete followed by recreate.
- Use only the field values present in the step or prior step results — never drop unaffected fields when recreating.

THE "hierarchical" INSTANCE NAME:
- "hierarchical" means the parent component itself, not a subcomponent.
- Use it only when the step explicitly says the endpoint is a top-level port of the parent component.
- If a step connects two subcomponent instances to each other, use their real instance names — never use "hierarchical" for a subcomponent endpoint.

ON FAILURE:
- If the tool returns an error, do NOT retry with the same or arbitrarily varied arguments.
- Only retry if the error message gives clear actionable information about what to correct.
- If the error does not give enough information to fix the call, report failure immediately.

After the tool call completes, write exactly one line:
Step result: <what was done and the key output value, e.g. an ID or status>
On failure, write exactly:
Step result: FAILED — <reason from the tool response>

> Do not write anything beyond that line.