

Increasing Resilience of Smart Home Agents

Christopher Terrazas, Eduardo Cotilla-Sanchez
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, Oregon, USA
{terrazac, ecs}@oregonstate.edu

Abstract—Smart homes and smart devices are becoming more prevalent across millions of homes around the world. With the rise of AI, the smart home industry is quickly increasing its integration to manage common smart home tasks. However, existing work in large language models (LLMs) as agents within smart homes have shown minimal resilience due to limited environment scenarios or poor performance in complex tasks. We explore several strategies for LLMs as agents within the popular open-source software (OSS) smart home automation framework HomeAssistant to increase overall smart home resilience. Our approach combines traditional supervised learning techniques and optimized prompting as the core learning process. We use a diverse set of LLMs covering different levels of reasoning and costs in our optimization pipeline and evaluate their performance on a subset of the hardest tasks in a smart home benchmark. We include ReAct and Reflexion agentic paradigms and reveal how both provide marginal return on investment compared to fine-tuned LLMs for multi-device control within HomeAssistant but show promise in resilience tasks such as failure response.

Index Terms—Agents, Edge Resilience, IoT, LLMs, Smart Home

I. INTRODUCTION

The smart home automation industry spans both commercial and OSS solutions. One of the issues with the latter is the user adoption of the software has a high learning curve and requires an in-depth level of technical expertise for setup and ongoing maintenance. This results in unsustainable use or total user abandonment for non-technical users. The burden extends to the developers of smart home frameworks due to its support of many diverse smart home devices [1]. LLMs as agents can help minimize the burden of smart home operational tasks, such as orchestrating smart device communication for a particular outcome. These types of tasks are synonymous with decision-making in the literature, where an agent decides an action within a particular environment given some current state.

There are a vast number of day-to-day tasks that an OSS smart home user can perform within their smart home. These include actions such as device control and interaction, powering on and off smart lights, analyzing grid power input and output usage, or adjusting comfort devices such as heat pumps and HVACs. In smart home environments, it is critical to not only accomplish a task correctly, but to also avoid creating mishaps, especially in high-critical spaces like smart buildings within the medical space, defense complexes, or computational research facilities.

Another common issue OSS smart home users face is troubleshooting. This process is similar to finding and triaging bugs within the field of software engineering. Troubleshooting for OSS smart home users consists of identifying problematic integrations, automations, or device conditions. For a new user, this can become overwhelming and difficult to sustain. For the OSS smart home user, the troubleshooting steps involve clicking their user interface to determine device status changes and deciphering error logs to pinpoint failure points. Internally, this is equivalent to the function of internal API endpoints within an OSS smart home framework. With the rapid improvements in popular LLMs, LLM-powered agents are more capable and can help a smart home user with troubleshooting tasks such as log parsing or reducing the speed of software development for engineers [2].

LLMs as agents have the ability to reason about their actions in their environment which make them suitable for exploration in the smart home space and as a tool for smart home users to increase their overall reliability and operational efficiency of their smart home.

II. ENVIRONMENTS

A. Home Assistant

HomeAssistant [3] has become the defacto OSS framework for smart home automation. It is written and maintained in Python, a popular programming language, that allows many hobbyists and programmers in the community to contribute to its development in a distributed fashion. However, HomeAssistant’s modular architecture and technical documentation can become overwhelming for both new and experienced users. There are many ways to install the software to support: mini computers, docker containers, and many different operating systems and hardware architecture combinations. After installing the software, users can begin to add devices and write automations.

We use HomeAssistant as a basis for testing and deploying our agents due to its stable nature in development and its design to separate varying levels of control. For example, HomeAssistant has authentication out of the box for users. This is a standard requirement for most smart home users and critical to restrict unauthorized access for smart homes integrated with LLMs and autonomous agents. The overall design of HomeAssistant consists of a hierarchical format starting with their Home Assistant Operating System, then

a Supervisor for managing the operating system, and finally the core software. For our resilience testing, we focus on the core software and its API endpoints to perform common smart home task failure responses.

III. RELATED WORK

A. Smart Home Agents

A common strategy for interacting with LLMs is the chain of thought [4] process where a user and LLM make progress through guided prompts by the user. In both Sage [5] and Sasha [6], prompts are used to train the agents. In Sasha, their approach focuses on action plans that rely on semantic context to make progress towards ambiguous goals from the user. In Sage, prompts are dynamically generated to guide the LLM throughout the action space.

Instead of creating methods to influence LLM behavior by constructing prompts as an emphasis, we adopt the Demonstrate-Search-Predict framework’s (DsPy) focus on programming through structured signatures and its built-in optimizers to preserve the agent’s training process and let DsPy handle the prompt optimization. We find this approach less sensitive to changes on the application layer and supports multiple functionalities similar to LangChain [7] within Sage. LangChain is an LLM framework that focuses on manual prompting and targets out of the box LLM development, but we chose DsPy because of its focus to support lower-level customization of LLM behaviors through modules. Another reason we chose to incorporate DsPy into our pipeline is because of the potential to optimize with existing machine learning techniques. For example, we configured our Reflexion agent with a custom heuristic for smart home tasks that we found to be pivotal in its performance.

Sasha and Sage both demonstrate portability in their approach by testing, designing, and evaluating different LLMs within their software to demonstrate LLM’s use of their particular technique. This is a crucial aspect to the scientific process, but it is also important for usability. For example, a user might switch LLMs within their smart homes or perhaps switch to a new smart home environment. We aim to take this step further and extend this flexibility to both the environment the smart home agent will operate in and the different scenarios within that smart home environment. In the latter, we focus on “deploying” these smart home agents within actual smart home environments in a way that is controlled and safe. For example, we discuss the guard rails we explored in section VI. We aim to slowly progress towards a production state where a smart home agent is actively managing a smart home environment within HomeAssistant. This is important as there can be high-stakes devices and automations within these smart home environments. In our experiment, we accomplish a hybrid version within a controlled environment with a combination of real devices and fake devices that have been configured for use for our agents.

IV. AGENT LLMs AND TESTING

The setup for our experiments consists of prompt optimization, transfer learning, evaluation, and deployment phases. The training and testing pipeline produces the agents we evaluate, and we further deploy these agents in a live HomeAssistant environment. Our focus is to demonstrate the performance of the LLMs while minimizing manual tasks and costs per token for each agent.

A. Large Language Models

We combine a mix of state of the art LLMs from both private industry and OSS. The cost for using private LLMs, especially agents, is significant [8]. To reduce the cost to train and test agents, we selected models that were cost efficient relative to industry pricing, but also performed well in reasoning tasks. To broaden the impact of our experiments, we chose to incorporate popular OSS LLMs as a core LLM engine in our experiments. For OSS models, we targeted medium to large LLMs because smaller models are broadly understood to struggle with agentic tasks [9].

From private industry, we used GPT-4 [10] as our teacher for bootstrapping. We then used GPT-5-mini [11], DeepSeek-R1 [12], and Grok-4-fast-reasoning [13]. From OSS models, we selected Qwen2.5 72B [14], Gemma-2-27B [15], and GPT-OSS-120b [16].

B. Benchmarks

1) *HomeBench*: We implement the HomeBench [17] benchmark which consists of 175,000 smart home test cases across five different categories: Valid single-device instruction, Invalid single-device instruction, Valid multi-device instruction, Invalid multi-device instruction, and Mix multi-device instruction (MM). In an effort to focus on complex tasks which agents are well-suited for, we selected the last category of multi-device instruction as the foundation for our analysis. We chose this category as it was the most difficult in HomeBench for all LLMs tested with a 38% success score for the best performing LLM. These types of tasks involve the agent interacting with multiple devices and tools to accomplish a specific expected output.

Once we identified the worst-performing category to test our agents, we extracted all of the 4,414 MM test cases. In each test case, there is an id, input, and expected output field. From the 4,414 MM test cases, we extracted 441 test cases at random as our initial training and validation datasets. We extended the original input field and added context for each training data entry, similar to what an agent would see during the agent decision-making phase in HomeBench. For example, the agent sees the agent system instructions, the specific task device lists and tools available along with their argument signatures, and finally the entire user request. We construct all of this as the training data for the agent to reduce the planning iteration space for an agent by learning from an example of what it would see in this specific smart home environment. We then use this data to begin our supervised learning strategy to increase overall performance of all agents we test. The subset

of data we created captures a diverse subset of the hardest category in the HomeBench dataset. The size of the dataset also reduces the cost of running experiments significantly.

2) *Resilience Dataset*: In an effort to increase resilience of agents in real smart home environments, we created a small resilience benchmark focused on failure recovery. Prior work has focused on common smart home tasks, but it does not prioritize more complex tasks. For example, there are smart home tasks that are more difficult to complete such as recovering failed automations, offline integrations, or monitoring and responding to abnormal behavior such as power spikes caused by the devices themselves or external factors such as power outages. The size of our resilience dataset is 50 test cases aiming to test our LLMs as agents' response to more difficult failure scenarios, which we hold 25 for training and testing. We used Claude Opus 4.7 [18] to generate our suite of failure response tests and manually verified each for correctness. The dataset is constructed among 10 distinct categories covering various scenarios a smart home user may encounter: power failure, integration outage, automation partial execution, cascading dependency, pre-event hardening (such as natural disasters), life safety, cloud outage, sensor disagreement, network partition, and update rollbacks.

V. LLM STRATEGIES

A. Base

For the base LLM strategy, we use each of the LLMs in IV-A and then load that specific model and run our evaluation with no modifications to the LLM. For example, this is equivalent to integrating a third-party LLM provider via user credentials and feeding queries to the LLM through the HTTP protocol via APIs. We chose to incorporate this as an initial baseline strategy among our various reasoning levels of LLMs within the smart home domain because of the common use among the smart home community.

For example, there are various third-party cloud LLM providers that HomeAssistant users have installed as an integration to use within their smart home. The base LLM strategy requires only creating and connecting the necessary credentials for each model provider. For a smart home user, they can install a particular cloud LLM integration and then provide these credentials to begin using the LLM within their HomeAssistant environment.

B. Optimized

The optimized strategy focuses on using DsPy's internal optimizers for a supervised learning approach to help all of our LLMs learn more about the smart home domain through training data. We explain and mention how we split this data within HomeBench and our resilience dataset in V-E. The optimized strategy is used to explore if using training data and additional context for an LLM can improve the smart home agent in various tasks. We chose to incorporate this LLM strategy due to the significant amount of smart home data provided within HomeBench.

C. ReAct

The ReAct paradigm interleaves an LLM within several distinct steps as its agent loop: reasoning, acting, and observing. It does this by using past traces of its actions and reasoning to help steer itself to a desirable outcome on its current task. The overall process is decomposed in an iterative manner, per decision iteration, so that the LLM can determine a grounded action based on its current trajectory and context.

In smart homes, this might be a common task such as finding a specific smart vacuum within a smart home and changing its status. This sounds trivial, but there may be several smart vacuums docked across different floors, and finding the correct one is the challenging part and not necessarily changing its status. In other more complex smart home tasks, such as troubleshooting, the LLM's observations drive its reasoning and actions to increase progress on a particular task. In this context, the action might be to change a smart device status, stop a problematic smart home automation that is currently running, or create a complex automation request from a user orchestrating several smart devices within the smart home.

We incorporate this popular agentic paradigm and focus its closed-loop reasoning system on the hardest smart home tasks within HomeBench and against our resilience dataset. When the ReAct agent receives a command from a user, it commences its feedback loop to accomplish these smart home tasks. We also chose to include ReAct to extend from the base and optimized LLM strategies, which focus accomplishing each smart home task in one attempt with no continuous reasoning or in a *zero-shot* manner.

D. Reflexion

To explore diverse types of agents, we implemented the Reflexion [19] agent as shown in Algorithm 1. This technique incorporates verbal reinforcement learning combined with self-reflections and long-term memory to assist the agent in complex tasks. The advantage to this approach is agent creators are able to utilize reinforcement learning strategies without the high compute costs of developing and training reinforcement learning models. The verbal portion acts as linguistic feedback to the agent on how it is doing in its current task episode based on the length of experiences stored in long-term memory. This is similar to how humans improve in difficult tasks by noting past failed experiences and adjusting to accomplish a particular goal. The authors showed that the Reflexion learning framework performed better than ReAct in complex tasks in the AlfWorld dataset [20].

The internal architecture for Reflexion is agnostic to the agent it uses. For example it implements the ReAct algorithm [21] for the self-improvement LLM that generates actions called the Actor as shown in Figure 1. We implemented this algorithm in a custom DsPy module. Within this module, we implemented the evaluation and self-reflection portions of Reflexion combined with the long-term memory for the Actor (the LLM). For our Actor, we kept the ReAct portion and combined it with Reflexion's self-reflecting process. For both the self-reflection and evaluator, we used DsPy's Predict

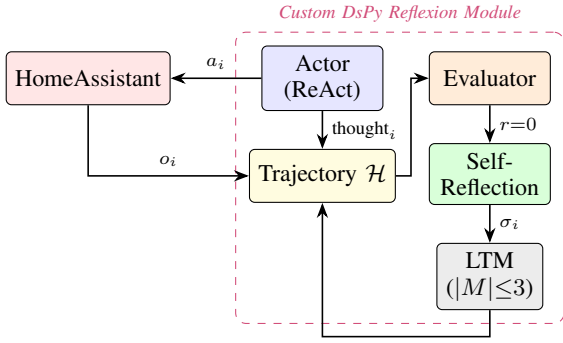


Fig. 1: Reflexion architecture. The Actor (ReAct) interacts with HomeAssistant and writes to a shared trajectory $\mathcal{H}$. After each action, the Evaluator scores $\mathcal{H}$; on a zero reward, the Self-Reflection module produces a verbal plan σ_i that is appended to a bounded FIFO long-term memory ($|M| \leq 3$). The contents of M are appended back into the trajectory at every step, making prior reflections visible to the Actor on subsequent decisions within the same episode. All four components are implemented as custom DsPy modules.

module to dynamically classify our desired output at each phase in the Reflexion algorithm.

In the self-reflection, we adopted the Reflexion authors’ fixed parameter size of three for the total experience length per episode to minimize the token context burden commonly faced in LLMs within agent tasks. In the evaluation portion, we implemented a metric inspired from the Reflexion Alfworld test case that returns a scalar reward signal r to the agent. During the evaluation portion, the Evaluator, another LLM created with DsPy, looks at the agent’s current trajectory thus far and checks for areas such as repeated actions contributing to the same errors. This portion is domain-specific for the agent’s environment.

E. Training

We benchmark several machine learning techniques to train and test the smart home agents powered by multiple large language models. One of the issues with implementing agents is the tokens generated are greater due to their agent loops. To minimize this burden, we conducted a focused experiment with optimization, supervised learning, and transfer learning techniques to boost agent performance. Note, we were able to fine-tune the OSS model weights with MLX [22] and showed signals of improvement, but we were unable to serve locally due to the size of the LLMs as explained in V-F.

1) *Optimization*: After extraction of the training data, we use the Demonstrate-Search-Predict framework [23] using DsPy to optimize our agents with this training data. These agents learn about the HomeBench environment, which was motivated from HomeAssistant and prior work. In this approach, we aim to increase the abilities of the agent prior to its interaction with unseen data as seen in traditional supervised learning strategies. This has the benefit of reducing the planning space or the time taken for the agent to learn the

Algorithm 1 Reflexion: Inline Verbal-RL Loop for Smart Home Agents

Require: Task inputs x , tools $\mathcal{T} \cup \{\text{FINISH}\}$, max iterations N

- 1: Initialize trajectory $\mathcal{H} \leftarrow \emptyset$
- 2: Initialize long-term memory $M \leftarrow \emptyset$ {bounded FIFO, $|M| \leq 3$ }
- 3: Initialize cumulative reward $r \leftarrow 1$
- 4: **for** $i = 1$ to N **do**
- 5: $(\text{thought}_i, a_i, \text{args}_i) \leftarrow \text{Actor}_{\text{ReAct}}(x, \mathcal{H})$
- 6: $o_i \leftarrow \text{Execute}(a_i, \text{args}_i)$ {Home Assistant API call}
- 7: Append $(\text{thought}_i, a_i, \text{args}_i, o_i)$ to $\mathcal{H}$
- 8: $r \leftarrow r \cdot \text{Evaluator}(\mathcal{H})$ { $\in \{0, 1\}$; checks confused thoughts, repeated errors}
- 9: **if** $r = 0$ **then**
- 10: $\sigma_i \leftarrow \text{SelfReflect}(r, \mathcal{H}, M)$ {verbal plan revising the strategy}
- 11: $M \leftarrow \text{Append}(M, \sigma_i)$ {evicts oldest if $|M| = 3$ }
- 12: **end if**
- 13: Append M to $\mathcal{H}$ as experience $_i$ {visible to Actor on next step}
- 14: **if** $a_i = \text{FINISH}$ **then**
- 15: **break**
- 16: **end if**
- 17: **end for**
- 18: $y \leftarrow \text{Extract}(\mathcal{H}, x)$ {ChainOfThought fallback to final outputs}
- 19: **return** $(y, \mathcal{H})$

basics of its environment. The key portion here is we do not have to perform manual prompting modifications which are brittle to future agent modifications.

We use DsPy’s Multiprompt Instruction PProposal Optimizer Version 2 (MIPROV2) [24] to automatically optimize the prompts and instructions for smart home tasks. We used DsPy’s built-in ReAct module and wrote a custom DsPy Reflexion module for our experiments. We annotated our smart home agent modules with the appropriate docstrings to explain the domain of smart homes and the agent’s responsibilities within the smart home environment.

2) *Supervised Learning*: For all of our experiments, we used HomeBench as a foundation for our smart home agents. To reduce costs and focus our strategy on agent evaluation, we used statistical power analysis and tested on a subset of the 175,000 HomeBench dataset. Our subset consisted of 4,414 smart home automation tasks from the worst performing category across the state of the art LLMs.

3) *Transfer Learning*: From this subset, we took 441 random examples from the original dataset to use as training data to optimize our best model on this task. After prompt optimization with MIPRO, we use transfer learning by having our best model act as a teacher for our less-capable LLMs. We use this strategy to reduce the overall costs of our experiments and to show techniques to squeeze agentic performance from

the best performing LLMs.

F. Testing

For the testing portion of our experiment, we sampled 250 random examples for a 95% confidence interval with a 6.2% margin of error. We evaluated each LLM under all four strategies (Base, Opt, ReAct, Reflexion) using this sample, then wrote a Python script to invoke the HomeBench test suite. For the private industry LLMs, we updated the configurations accordingly to include the relevant API keys and API base urls per the OpenAI standard [25].

For the OSS LLMs, we used OpenRouter [26], an OSS LLM hosting provider for inference. We attempted to self-host with Ollama [27], but ran into memory issues during inference. We explored other viable solutions such as vLLM [28], a high-throughput and memory-efficient inference and serving engine for LLMs which might be more suitable for running larger models locally. However, our machine was not compatible with the vLLM software.

VI. DEPLOYMENT

A. Guardrails

To ensure the agent had limited access in its scope, we restricted its available tools to the HomeAssistant Rest API. This API consists of a set of endpoints designed to manage connected smart devices, integrations, and automations. HomeAssistant supports LLM interactions with the Home Assist API, but we chose not to use this because the scope was very limited and excluded all administrative tasks for the LLM. For example, an LLM is required to interact with this API through intents over exposed entities. To achieve finer-grain control over the aspects of the smart home and to give our agents greater autonomy, we chose our custom approach of integrating with the HomeAssistant Rest API. For each tool, we also included additional context for the smart home agent through annotated docstrings that detailed crucial information such as the tool name, expected number of parameters, and the expected output.

For security, we created a temporary token that the agent must use for authorization for each of its tool endpoints. The token can also be revoked manually by an authorized user within the HomeAssistant settings dashboard before its expiration. Our goal here was to simulate a privileged user’s access within the smart home, and the ability to revoke and grant access instantly for the smart home user. We did not give the agent full access to all of the HomeAssistant APIs including the Hypervisor to remove any interaction with the operating system and network configurations. This approach reduces the ability of the smart home agent to execute undesirable actions that might arise from hallucinations or prompt injections by manually writing and configuring its tools.

For the deployment environment, we configured the HomeAssistant dashboard with a hybrid approach of dummy devices and real smart devices such as a smart light. However, we did not include any critical devices in the smart home to minimize

the overall risk level of the agent and the smart home devices themselves.

B. Agent Interaction

Once deployed, we were able to interact with the agent via code and feed it queries to fetch information such as the current weather from the weather integration, statuses of various smart home devices, and it was able to generate working automations written in YAML. We were able to feed the agent natural language commands to successfully turn on and off smart lights. We also tried to access unauthorized endpoints within the Hypervisor and were unable to access any of the restricted areas via the agent commands. However, during this process, the agent would attempt to fetch the desired information from our prompts with tools it did have access to, but could not.

VII. EVALUATION

TABLE I: F1 Evaluation on HomeBench MultiDevice Tasks (%)

Model Name	LLM Strategy			
	Base	Opt	ReAct	Reflexion
gpt-oss-120b	86.7	88.6	85.4	85.0
gpt-5-mini	86.2	86.2	85.3	84.7
grok-4-1	86.3	89.9	88.0	87.8
deepseek-r1	79.0	85.4	77.7	76.6
qwen-2.5-72b	46.4	52.4	45.4	44.6
gemma-2-27b	34.3	37.7	27.7	28.8

Sample-averaged F1 over $n=250$ scenarios sampled from 4,000 (95% confidence, $\pm 6.2\%$ worst-case margin).

We compute the F1 score similarly to HomeBench and compare predicted commands with a gold standard, then we calculate the unweighted mean across F1 scores for all of our examples. Our results show that optimized prompting and zero-shot LLMs are a good foundation for accomplishing complicated multi-device smart home tasks as seen in Table 1. For most of the LLMs, the addition of agents degraded their performance compared to their base and opt level performance. The greatest disparity we observed was gemma-2-27b, with a drop in F1 score by 19.2% in ReAct and a 16% drop in Reflexion. This further supports that smaller models struggle in agentic reasoning loops.

However, it is worth noting that the other LLMs showed minimal improvements by adding ReAct and Reflexion agents. The only LLM to show an improvement from its base performance in both ReAct and Reflexion was grok-4-1 with a marginal 2% increase.

For our resilience dataset, we see a different pattern emerge where agents are generally performing as good or better than zero-shot LLMs as seen in Table 2.

VIII. DISCUSSION AND CONCLUSION

We showed ReAct and Reflexion’s performance compared to several different optimized LLMs on a subset of the

TABLE II: F1 Evaluation on Resilience Dataset (%)

Model Name	LLM Strategy			
	Base	Opt	ReAct	Reflexion
gpt-oss-120b	43.2 $\pm$ 8.2	45.2 $\pm$ 8.8	42.3 $\pm$ 8.7	47.1 $\pm$ 10.2
gpt-5-mini	43.8 $\pm$ 8.6	45.0 $\pm$ 8.6	44.7 $\pm$ 7.9	44.4 $\pm$ 8.8
grok-4-1	47.0 $\pm$ 9.6	47.7 $\pm$ 9.3	48.6 $\pm$ 10.0	46.1 $\pm$ 9.4
deepseek-r1	43.7 $\pm$ 8.4	45.0 $\pm$ 9.1	46.6 $\pm$ 9.1	41.1 $\pm$ 8.3
qwen-2.5-72b	41.2 $\pm$ 8.0	41.2 $\pm$ 8.0	41.3 $\pm$ 7.9	41.6 $\pm$ 8.4
gemma-2-27b	41.2 $\pm$ 7.5	35.3 $\pm$ 5.3	37.6 $\pm$ 5.1	33.2 $\pm$ 7.0

Values are mean F1 $\pm$ 95% confidence t-interval ($n=25$).

HomeBench dataset. Our results show that optimized prompting and zero-shot LLMs are a good foundation for accomplishing complicated multi-device smart home tasks. In an effort to better leverage agents in smart homes, we created a small resilience benchmark where agentic strategies as a whole outperformed baseline LLMs on resilience tasks. This was opposite of the trend we saw in multi-device smart home tasks, where agents under-performed for most models. In an actual smart home environment, we recommend a hybrid approach where zero-shot LLMs can handle multi-device smart home tasks and have agents for failure response tasks.

Future work can explore the creation of more complex smart home benchmarks where agentic paradigms may perform better as in our failure response focus. Furthermore, exploration is needed in the reduction of overall training and computation costs of interacting with agents to minimize the token consumption of the planning process. For agents, this cost grows by a considerable factor compared to zero-shot LLMs. We recommend techniques similar to our strategy of using transfer learning and optimization techniques to boost overall agent performance. With this strategy, we were able to push OSS LLMs performance in the HomeBench’s most difficult domain by a significant margin. Our approach can be used to reduce inference costs further by self-hosting a fine-tuned OSS LLM, which many HomeAssistant users opt in for [29]. The self-hosting approach also increases smart home resilience by removing the dependence on the internet for full smart home agent operation. We acknowledge the difficulty and sustainability of this specific approach as it requires constructing a dataset for specific domain tasks. However, small datasets can be instrumental in improving performance using zero-shot techniques.

REFERENCES

- [1] D. Rodríguez-Cárdenas, S. A. Khan, P. Mandal, A. Nadkarni, K. Moran, and D. Poshyany, “Testing Practices, Challenges, and Developer Perspectives in Open-Source IoT Platforms,” in IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 290–301, 2025.
- [2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., “Evaluating Large Language Models Trained on Code,” arXiv:2107.03374, 2021.
- [3] Home Assistant, “Home Assistant: Open source home automation that puts local control and privacy first,” GitHub repository, 2013. [Online]. Available: <https://github.com/home-assistant/core>
- [4] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models,” in Proc. 36th International Conference on Neural Information Processing Systems (NeurIPS), New Orleans, LA, USA, pp. 24824–24837, 2022.

- [5] D. Rivkin, F. Hogan, A. Feriani, A. Konar, A. Sigal, S. Liu, and G. Dudek, “SAGE: Smart Home Agent with Grounded Execution,” arXiv:2311.00772, 2023.
- [6] E. King, H. Yu, S. Lee, and C. Julien, “Sasha: Creative Goal-Oriented Reasoning in Smart Homes with Large Language Models,” Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. (IMWUT), vol. 8, no. 1, Article 12, pp. 1–38, Mar. 2024.
- [7] H. Chase, “LangChain,” GitHub repository, 2022. [Online]. Available: <https://github.com/langchain-ai/langchain>
- [8] J. Wang, S. Jain, D. Zhang, B. Ray, V. Kumar, and B. Athiwaratkun, “Reasoning in Token Economies: Budget-Aware Evaluation of LLM Reasoning Strategies,” in Proc. 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP), Miami, FL, USA, pp. 19916–19939, 2024.
- [9] X. Liu, H. Yu, H. Zhang, Y. Xu, X. Lei, H. Lai, Y. Gu, H. Ding, K. Men, K. Yang, S. Zhang, X. Deng, A. Zeng, Z. Du, C. Zhang, S. Shen, T. Zhang, Y. Su, H. Sun, M. Huang, Y. Dong, and J. Tang, “AgentBench: Evaluating LLMs as Agents,” in Proc. 12th International Conference on Learning Representations (ICLR), 2024.
- [10] OpenAI, “GPT-4 Technical Report,” arXiv:2303.08774, 2023.
- [11] OpenAI, “GPT-5 System Card,” arXiv:2601.03267, 2025.
- [12] DeepSeek-AI, “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning,” arXiv:2501.12948, 2025.
- [13] xAI, “Grok 4 Fast Model Card,” Technical Report, Sep. 2025. [Online]. Available: <https://data.x.ai/2025-09-19-grok-4-fast-model-card.pdf>
- [14] Qwen Team, “Qwen2.5 Technical Report,” arXiv:2412.15115, 2024.
- [15] Gemma Team, Google DeepMind, “Gemma 2: Improving Open Language Models at a Practical Size,” arXiv:2408.00118, 2024.
- [16] OpenAI, “gpt-oss-120b & gpt-oss-20b Model Card,” arXiv:2508.10925, 2025.
- [17] S. Li, Y. Guo, J. Yao, Z. Liu, and H. Wang, “HomeBench: Evaluating LLMs in Smart Homes with Valid and Invalid Instructions Across Single and Multiple Devices,” in Proc. 63rd Annual Meeting of the Association for Computational Linguistics (ACL), 2025.
- [18] Anthropic, “Claude Opus 4.7,” 2026. [Online]. Available: <https://www.anthropic.com/claude/opus>
- [19] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, “Reflexion: Language Agents with Verbal Reinforcement Learning,” in Proc. 37th International Conference on Neural Information Processing Systems (NeurIPS), Red Hook, NY, USA, pp. 8634–8652, 2023.
- [20] Shridhar, M., Yuan, X., Côté, M.-A., Bisk, Y., Trischler, A., and Hausknecht, M., “ALFWorld: Aligning Text and Embodied Environments for Interactive Learning,” in Proc. 9th International Conference on Learning Representations (ICLR), 2021.
- [21] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, “ReAct: Synergizing Reasoning and Acting in Language Models,” arXiv:2210.03629, 2023.
- [22] A. Hannun, J. Digani, A. Katharopoulos, and R. Collobert, “MLX: Efficient and Flexible Machine Learning on Apple Silicon,” 2023. [Online]. Available: <https://github.com/ml-explore/mlx>.
- [23] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts, “DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines,” in Proc. 12th International Conference on Learning Representations (ICLR), 2024.
- [24] K. Opsahl-Ong, M. J. Ryan, J. Purtell, D. Broman, C. Potts, M. Zaharia, and O. Khattab, “Optimizing Instructions and Demonstrations for Multi-Stage Language Model Programs,” in Proc. 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2024.
- [25] OpenAI, “OpenAI API Reference,” [Online]. Available: <https://platform.openai.com/docs/api-reference>
- [26] OpenRouter, “OpenRouter: A Unified Interface for LLMs,” OpenRouter, Inc. [Online]. Available: <https://openrouter.ai/>. [Accessed: Jun. 25, 2026].
- [27] Ollama, “Ollama: Get up and running with large language models locally,” Ollama. [Online]. Available: <https://ollama.com/>. [Accessed: Jun. 25, 2026].
- [28] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, “Efficient Memory Management for Large Language Model Serving with PagedAttention,” in Proc. 29th ACM Symposium on Operating Systems Principles (SOSP), pp. 611–626, 2023.
- [29] Home Assistant, “Home Assistant Analytics,” [Online]. Available: <https://analytics.home-assistant.io/>.