Universiteit Leiden

Cloud vs. Local Multimodal Models for Embodied Task Planning: Evaluating Trade-offs in Success, Latency, and Energy Consumption

Péter Benke (s4074300)

Supervisors:
Aske Plaat & Marijn Prins

Bachelor Thesis– Data Science and Artificial Intelligence

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

July 7, 2026

Abstract

This thesis evaluates whether locally run open-weight language and multimodal models can provide a practical alternative to cloud-based models for high-level embodied task planning. The evaluation is performed in ALFWorld using a controlled six-task household benchmark in text-only and visual observation modes. Models interact with the environment through a strict tool-calling interface, where each response must produce exactly one executable action. Across 1020 completed episodes, 816 succeeded, giving an overall success rate of 80.0%. The results show that several local models were competitive with the API-hosted comparison models, but that performance depended strongly on the selected model and serving setup. Although `Qwen3.6-27B` achieved the highest success rate, `Qwen3.6-35B-A3B` provided the strongest overall balance between success, latency and measured local GPU energy. Visual input did not improve success in this benchmark setup and consistently increased inference latency.

Contents

1 Introduction

1.1 Motivation

Robotics systems increasingly need to operate in environments where tasks are specified through natural language rather than fixed commands. In domestic service and assistive settings, a robot may be asked to complete high-level goals such as finding an object, moving it to a location, or interacting with several objects in sequence. Such tasks are highly context specific in their nature, involving variables such as current state of the robot, available objects, possible constraints and the user's instructions. This makes it difficult to reduce these tasks to a fixed list of predefined commands. As a result, these systems require not only perception and control, but also the ability to interpret instructions, reason about goals, and select appropriate actions over multiple steps.

Large language models and multimodal models have become attractive components for such systems because they can support high-level reasoning and task planning [8]. Instead of relying only on manually specified rules or rigid task-specific pipelines, a language model can be given a task instruction with a representation of the current environment, after which it can break it down into sub-tasks or select the next action. This makes such models highly relevant for robotics tasks in which the agent must repeatedly decide what to do next based on a continuously changing environment. In this role, the model does not necessarily replace the entire robotic system, but it can function as a reasoning or planning component within a larger perception-action loop.

However, the strongest models are often computationally expensive and commonly accessed through cloud-based services. This creates practical limitations for robotics. First, the total latency of the system is not only affected by the inference time of the model itself, but also by networking latency and external service conditions. Second, operation is entirely dependent on internet availability and the service availability of the language model provider itself, which can be problematic for systems that are expected to function reliably with no downtime.[1] Third, cloud-based inference may raise privacy concerns when observations, user instructions or environmental data are transmitted to external servers. Finally, the deployer of the robotics service has limited control over changes to the model, the serving infrastructure and the exact computational resources used during inference.

Lightweight open-weight models that run locally offer a possible alternative. Local inference reduces dependency on cloud infrastructure and gives the deployer significantly greater control over model version, deployment conditions, inference settings and data handling. It also makes it possible to directly measure the resource use of the hardware on which the model is run. These properties are particularly relevant for robotics, where deployment may be constrained by available hardware, response time requirements and energy limitations. Locally deployed models may therefore be practically attractive, even if they are not the most capable models in terms of raw reasoning performance.

[1] It is entirely dependent only with the assumption that the robot cannot function without its language model component.

1.2 Problem Statement

Replacing a cloud-based model with a smaller local model may significantly reduce task performance. Smaller models may produce less reliable plans, misunderstand task constraints, repeat ineffective actions, or generate outputs that do not correspond to valid actions within the current conditions of the environment. In robotics-oriented tasks, failures like these can be especially important because a single incorrect decision may prevent the completion of a longer task sequence. Therefore, the practical value of local models cannot be determined by efficiency alone: it must be evaluated in relation to their ability to complete the task successfully.

This creates a trade-off between task success, inference latency and energy consumption. A large cloud-based model may provide stronger reasoning, but can introduce external dependencies, and additional ongoing costs. A smaller local model may respond faster or use less (locally measurable) energy, but may also fail more often. For robotics systems, the most useful model is therefore not necessarily the one with the highest task success rate in isolation, but the one that provides an appropriate balance between reliability, responsiveness and resource efficiency.

This trade-off between task success, inference latency and energy consumption remains insufficiently understood for embodied task planning systems. In particular, it remains unclear how much task performance is lost when replacing large cloud-based models with lightweight locally run open-weight alternatives, and whether potential gains in latency, deployability and local energy consumption compensate for that loss.

Answering this question in a general sense would require evaluation across many tasks, environments and robotic systems. Such a broad evaluation is beyond the scope of this thesis. Instead, this thesis investigates the trade-off within the ALFWorld [21] simulated embodied task benchmark that provides a controlled setting for evaluating sequential decision-making in household-like tasks. The focus is therefore on high-level task planning and action selection, rather than on physical robot control, low-level manipulation, or real-world perception. Within this scope, model replacement is evaluated not as a purely accuracy-based question, but as a practical trade-off between task success, inference latency, and local energy consumption.

1.3 Research Question and sub-questions

The main research question of this thesis is:

> How does replacing large, cloud-based language and/or multimodal models with lightweight, locally run open-weight variants affect the trade-off between task success, inference latency and energy consumption in ALFWorld embodied task environments?

With the following sub-questions:

- **SQ1:** How do cloud-based and locally run models compare in task success rate on ALFWorld tasks?
- **SQ2:** How do the evaluated models differ in inference latency per action and total inference time per episode?

- **SQ3:** How much energy is consumed during local inference, and how does this vary across locally run model configurations?
- **SQ4:** What trade-offs can be observed between task success, inference latency and energy consumption?

1.4 Thesis Overview

This bachelor thesis was carried out at LIACS under the supervision of Aske Plaat and Marijn Prins.

Section 2 provides the necessary background. Section 3 discusses related work. Section 4 describes the experimental design and measurement procedure. Section 5 explains the benchmark implementation.Section 6 provides the GitHub repository used for reproducibility. Section 7 presents the experimental results. Section 8 discusses the findings and their implications. Finally, Section 9 summarizes the main conclusions.

2 Background

2.1 Embodied AI and Task Planning

Embodied AI concerns agents that interact with an environment through perception and action. In this setting, an agent receives observations, selects actions and uses the resulting feedback from the environment to continue the task. Simulated environments are commonly used in embodied AI research, because they provide a controlled virtual testing setting for training and evaluating agents before deployment in physical environments [3]. [2]

Task planning in embodied environments is sequential. A task such as finding, moving or interacting with an object cannot usually be solved in one step. The agent must decide which action to take next, execute it, verify success, update its understanding after feedback and continue until the goal is reached. This process makes errors cumulative: a wrong action still affects the state of the agent, which can lead to inefficient behavior invalid states or failure to complete the task.

For this thesis, the relevant level is high-level task planning rather than low-level motor control. The evaluated models select abstract actions in ALFWorld, they do not control motors, calculate trajectories, or use real sensors. Since each episode requires repeated decisions, the model affects not only task success but also total inference latency and local energy consumption.

2.2 Language and Multimodal Models

Language models have long been studied in natural language processing, but the recent development of large language models is closely connected to advances in neural model architectures and large-scale training [17]. Earlier neural models, such as recurrent neural networks, processed tokens sequentially, which made large-scale parallel training more difficult and limited their ability to model long-range dependencies efficiently. The transformer replaced recurrence with an attention-based

[2] Some sources also refer to simulated embodied AI as disembodied AI [14].

architecture, enabling more parallelizable training and making it practical to train substantially larger language models [23]. Subsequent work showed that scaling up large language models can lead to strong performance across a wide range of tasks, including zero-shot and few-shot settings, where a model performs a task from instructions or a small number of examples without task specific fine-tuning [1]. These developments led to the emergence of large language models as general-purpose text generation systems that can be used for tasks such as reasoning, instruction following and planning [8].

The development of large language models involves many topics, including architecture design, pre-training, alignment, instruction tuning and scaling. These topics are not the main focus of this thesis. The discussion below is therefore limited to model properties that are relevant for the experimental comparison: modality, openness, model size, quantization and sparse activation.

2.2.1 Multimodality

Multimodal models extend language models by allowing generation to be conditioned on additional input modalities. In this thesis, the relevant case is vision-language modelling, where a model receives both textual and visual information and produces textual output. This is important for embodied task settings because an agent may need to select actions based not only on a task instruction or textual information, but also on the current visual state of the environment. Since language models operate over token-based representations, visual inputs must first be encoded into a form that can be combined with textual information. Modern vision-language models commonly address this by connecting a visual encoder with a language model, allowing generated text to be conditioned both on image and text inputs [12].

2.2.2 Numerical Precision and Quantization

Numerical precision describes the format used to store and compute model values, such as weights and activations. While model size is often described by parameter count, the memory required to store a model also depends on the number of bits used per parameter. For example, a model stored in 16-bit precision requires substantially more memory than the same model stored using 8-bit or 4-bit representations. This distinction is important for local-inference, where the available memory can determine whether a model can be executed at all.

Quantization reduces the numerical precision of model values, most commonly the weights, in order to lower memory requirements and make inference more efficient. In the context of this thesis, this is especially relevant because the locally executed models are run through llama.cpp [5] using GGUF model files. llama.cpp provides its own GGUF quantization formats, such as 8-bit 5-bit, and 4-bit variants, rather than relying on a single general quantization method. These formats reduce the memory footprint of the model and may improve inference speed, although the actual effect depends on the model, quantization level, hardware and inference backend [6].

Quantization can also affect model behavior. Lower precision introduces approximation error, which may reduce output quality or make the model less reliable in some settings. For ALFWorld, such degradation could appear as invalid actions, weaker instruction following or less consistent task planning. Quantization is therefore treated as part of the model configuration in this thesis.

2.2.3 Mixture-of-Experts Models (MoE)

Not all large language models use a dense architecture. In dense transformer models, each token is processed by the same set of model layers and parameters. Mixture-of-Experts (MoE) models instead divide part of the network into multiple expert modules and use a router to select only some of these experts for each token [19, 4]. This allows MoE models to increase their total number of parameters without requiring all parameters to be active for every generated token.

This makes model size difficult to interpret. For dense models, parameter count is a relatively straightforward description of model scale. For MoE models, a distinction has to be made between total parameters and active parameters. Total parameters describe the full model that must be stored, while active parameters describe the part of the model used for the computation of a token. This distinction is used in modern open-weight MoE models such as Mixtral, where the reported total parameter count is substantially larger than the number of active parameters used during inference [11].

For local inference, this distinction matters because total parameters affect memory requirements, while active parameters are more closely related to the computation performed during generation. MoE architecture is therefore treated as part of the model configuration when comparing task success, inference latency and local energy consumption.

2.3 Deployment Settings

Model inference can be deployed in different settings depending on where the model is executed and which infrastructure is available. For robotics-oriented systems, this choice affects latency, reliability, privacy, hardware requirements and energy consumption. This thesis distinguishes between three deployment settings: Cloud-Based Inference, Local-Server Inference and On-Device Inference.

2.3.1 Cloud-Based Inference

In cloud-based inference, the model is executed on infrastructure operated by an external provider and accessed through an API. This can provide access to strong models without requiring the user to own or maintain the necessary hardware. However, the total response time includes not only model inference, but also network communication and provider-side serving conditions. Cloud-based inference also introduces dependencies on internet connectivity and provider availability, and it may raise privacy concerns when user instructions or environmental observations are transmitted to external servers.

The energy consumption of cloud infrastructure is an important broader issue, especially as AI workloads contribute to increasing data-center electricity demand [10]. However, this thesis does not attempt to estimate the environmental impact or server-side energy consumption of cloud inference. Instead, cloud-based models are evaluated primarily in terms of task success and end-to-end latency. In this sense, cloud side energy consumption is outside the measurement scope of this thesis, although API usage costs remain a practical deployment consideration.

2.3.2 Local-Server Inference

In local-server inference, the model is executed on hardware controlled by the user or institution, such as a workstation or local server, while the robot communicates with this machine over a network. This deployment setting reduces dependence on external cloud providers and allows greater control over model version, inference settings, hardware configuration and data handling. It also makes local energy consumption directly measurable. Compared with On-Device Inference, a local server can use more powerful hardware and is less constrained by the robot's battery capacity, size and cooling limitations. However, this setting still introduces power costs, heat generation and possible fan noise at the server side. It also adds a communication path between the robot and the local server, so latency depends on both inference latency and network delay. In a controlled LAN network, this communication can be relatively stable, while WAN or mobile-network deployment may introduce greater latency variation and reliability concerns.

2.3.3 On-Device Inference

In On-device Inference, the model runs directly on the robot's onboard hardware. This avoids dependence on an external server and can reduce privacy and security risks because observations do not need to leave the robot for model execution. It can also remove network communication from the inference loop. However, this is the most constrained deployment setting. The model must fit within the robot's available compute, memory, storage, cooling and power budget. If the robot is battery-powered, energy consumption directly affects operating time. Power draw is also relevant because it produces heat and may require active cooling, which can increase fan noise, size and mechanical complexity. For this reason, on-device inference places stronger pressure on model size, quantization, and inference efficiency than Local-Server Inference.

2.3.4 Deployment Trade-offs

The three deployment settings therefore imply different interpretations of latency and energy. Cloud-based inference shifts most hardware and energy considerations to the provider, but introduces API costs and external dependencies. Local-server inference makes hardware use and electricity consumption measurable, but adds a communication path between robot and server. On-device inference is the most self-contained setting, but also the most constrained because power draw directly affects battery life, heat, cooling, and noise. These distinctions are important when interpreting the experimental results of this thesis, since the practical value of a model depends not only on task success, but also on the deployment setting in which the model is expected to operate. Table 1 summarizes the main differences between the three discussed deployment settings for model inference.

2.4 The ALFWorld Benchmark

ALFWorld is an interactive benchmark designed to align text-based reasoning with embodied household task environments. It builds on TextWorld [2], which provides interactive text-based environments and ALFRED [20], which defines household tasks in simulated embodied environments. The purpose of ALFWorld is to connect high-level language-based task reasoning with embodied

Factor	Cloud-based	Local server	On-device
Available compute	High	Medium–High	Low
Network dependency	High	Medium	Low–None
Latency variability	High	Low–medium	Low
Direct energy relevance	Minimal	Medium	High
Power/thermal constraints	None	Medium	High
Battery relevance	None	None	High
Cost structure	API usage	Hardware + electricity	Hardware + runtime limits
Privacy/control	Low	Medium–High	High

Table 1: Qualitative comparison of deployment settings for model inference.

task goals, making it suitable for studying agents that must choose actions over multiple steps rather than produce a single isolated response [21].

This benchmark contains household tasks derived from ALFRED, including Pick & Place, Look in Light, Clean & Place, Heat & Place, Cool & Place, and Pick Two & Place. These tasks require the agent to find objects, inspect or manipulate them, and place them in target receptacles or locations. Interaction in ALFWorld is expressed through high-level textual commands, such as moving a receptacle, opening or closing a receptacle, taking an object, placing an object, or applying task-specific actions such as cleaning, heating, or cooling. A task is completed only when the required goal conditions are satisfied.

This high-level action interface is important for interpreting the benchmark in this thesis. The evaluated model is not a complete robotic controller and does not directly control low-level motion, grasping, camera movement or physical actuation. Instead, it acts as a high-level decision making component: given the current observation and the task goal, it must select the next action. In visual mode, the model additionally receives an image observation from the simulated environment, but the action interface remains at the same high-level abstraction.

ALFWorld is therefore relevant because it provides a controlled setting for evaluating sequential embodied task planning. Since tasks require multiple observation-action steps, success depends on the model's ability to maintain a goal-directed behavior across a sequence of decisions. As a simulated and abstracted environment, ALFWorld should be viewed as a benchmark for high-level task planning rather than full physical robot deployment.

3 Related Work

This thesis builds on prior work in embodied task benchmarks, language-model agents, and efficient model inference. ALFRED introduced a benchmark for completing household tasks from natural-language instructions and first-person visual observations in simulated environments [20]. ALFWorld extends ALFRED by providing a text-based counterpart for each benchmark task, mapping embodied household activities into an interactive text environment where agents act using textual observations and commands while pursuing the same underlying goals [21]. Although these benchmarks were meant to study agents with different architectures, they are directly relevant as my thesis builds on top of them.

Large language models have also been studied as planning and decision-making components for interactive environments. ReAct interleaves reasoning and acting, allowing a model to update its decision based on environment feedback and evaluates this approach on tasks including ALFWorld [24]. AgentBench evaluates LLMs across several interactive environments, including ALFWorld and compares the agent abilities of strong commercial models and open-source alternatives [13]. These works are closely related to the task-success aspect of this thesis, but their main focus is agent reasoning and interaction performance rather than deployment oriented trade-offs.

Research has also examined the efficiency and resource requirements of language model inference. Samsi et al. evaluate inference performance and energy cost for LLaMA models across different GPU configurations [18]. Luccioni et al. compare the energy and carbon costs of different machine learning systems and show that generative models can be substantially more expensive than task-specific systems during inference [15]. Work on edge and local-cloud inference further shows that deployment choices affect model quality, latency, energy use and cost: Husom et al. study quantized LLM inference on edge hardware, while Yuan et al. examine when inference should be performed locally or offloaded to the cloud [9, 25].

Existing work therefore addresses several parts of the problem studied in this thesis: embodied household benchmarks, LLM-based agents, and efficient or energy-aware inference. However, these directions are usually studied separately. This thesis combines them by evaluating cloud-based and locally run multimodal models on ALFWorld tasks while measuring task success, inference latency and local energy consumption.

4 Methodology

This section describes how the ALFWorld benchmark was configured, executed, and analyzed.

4.1 Experimental Design

The experiment compares model behavior in two observation modes. In text mode, the model receives only the textual ALFWorld observation. In visual mode, the model receives the same textual observation together with a rendered image frame from the current AI2-THOR view. The action space remains identical in both modes. This makes it possible to compare whether the additional visual input changes task success, latency and resource use while keeping the underlying task and command interface fixed.

A central design choice in this benchmark is the use of a structured tool-call action interface. This interface is introduced for this thesis and is not part of the standard ALFWorld setup. In standard ALFWorld interaction, an agent submits textual commands such as `go to cabinet 1` directly to the environment. Early experiments showed that this free-text command format was fragile for smaller models: responses sometimes included additional reasoning text, multiple commands, malformed commands or slightly invalid variants of otherwise appropriate actions. These failures made it difficult to separate task-planning failures from output-format failures.

To make action parsing more consistent across models, the benchmark requires each action to be expressed as an OpenAI-compatible tool call. The models used in this benchmark are trained

to support structured tool calling, which makes this interface a natural and appropriate way to constrain their outputs. The tools closely mirror ALFWorld's natural-language command templates. For example, the tool `go_to` takes a `receptacle` argument and is deterministically converted into the textual ALFWorld command `go to <receptacle>`. All other valid actions are converted in the same way to their corresponding ALFWorld command strings.

The tool calls do not invoke separate ALFWorld APIs and do not provide additional environment capabilities. They are only an output-structuring layer. After a valid tool call is received, the benchmark translates it back into a plain-text ALFWorld oracle command and submits that command to the environment. The underlying action semantics are therefore still ALFWorld action semantics. The custom interface changes how the model expresses an action, not what actions are available.

For valid action-producing turns, execution proceeds as follows:

Model response: checked for exactly one valid tool call.
Tool call: translated into one plain-text ALFWorld command.
Environment step: executes the translated command in ALFWorld.

This chain applies only after the model response has passed tool-call validation. If a response contains no tool call, multiple tool calls, an unknown tool name, invalid arguments or otherwise malformed tool-call output, it is logged as an invalid model response. Such responses count as model turns and consume time and tokens, but they are not translated into ALFWorld commands and do not advance the environment. Consequently, the benchmark distinguishes between model turns and environment steps: `turn_count` records all model attempts, while `step_count` records only valid actions executed in ALFWorld.

Each benchmark episode consists of one model attempting a single ALFWorld task. The environment resets at the start, after which the model iteratively receives observations and selects actions. An episode terminates under the following conditions[3]:

- The task is successfully completed.
- The maximum number of environment steps is reached.
- The model call times out.
- Too many invalid tool responses occur consecutively.

The final dataset consists of both locally executed models and externally served API models. Models are evaluated in text and visual mode, except for one API-hosted model that does not support image input and is therefore evaluated only in text mode. Each model/mode combination contains 60 episodes. In total, the dataset contains 1020 completed episodes: 840 local-model episodes and 180 API-model episodes.

[3]Technically there was another termination condition in which the environment ends without success, but that was only there as a fallback in the case environment ends, but the `won` variable is `False`. This never happened.

$$\text{Total Dataset: } 1020 = \Bigg(\underbrace{7 \times 2}_{\text{local}} + \underbrace{2 + 1}_{\text{cloud}}\Bigg) \times \underbrace{(6 \times 10)}_{\text{60 episodes}} = \underbrace{540}_{\text{Text (9)}} + \underbrace{480}_{\text{Visual (8)}}$$

The full dataset is used for success, stop-reason, interaction-count and latency analysis. Energy analysis is restricted to the local model set.

4.2 Model Selection

The evaluated models were selected to represent recent locally deployable language and multimodal models, supplemented by externally served API models for comparison. Recency was used as a selection criterion because the aim was to evaluate models representative of current local and API-hosted deployment options. Within the local model set, the selection also aimed to cover a practical range of model scales, from smaller models such as `Gemma-4-E4B` to larger models such as `Gemma-4-31B`. In this context, *large* refers to the largest models that could still be served on the available hardware, as discussed further in Hardware setup and Appendix A.

Provider	Model	Quantization	GPU Setup	Modes	Energy measured
Local	`Gemma-4-E4B`	Q8_0	Single	Text, visual	Yes
Local	`Qwen3.5-9B`	Q8_0	Single	Text, visual	Yes
Local	`Gemma-4-12B`	UD-Q4_K_XL	Single	Text, visual	Yes
Local	`Gemma-4-26B-A4B`	UD-Q4_K_XL	Dual	Text, visual	Yes
Local	`Qwen3.6-27B`	UD-Q4_K_XL	Dual	Text, visual	Yes
Local	`Gemma-4-31B`	UD-Q4_K_XL	Dual	Text, visual	Yes
Local	`Qwen3.6-35B-A3B`	UD-Q4_K_XL	Dual	Text, visual	Yes
Cloud/API	`GPT-5.4-Nano`	N/A	N/A	Text, visual	No
Cloud/API	`DeepSeek-V4-Flash`	FP8[4]	N/A	Text	No

Table 2: Evaluated model and mode combinations. Local models are listed in ascending order parameter count. Quantization and GPU setup are part of the evaluated local deployment configuration.

The API models are included as externally served comparison points. They provide a reference for model behavior under the same task structure, while their provider-side serving configuration remains outside the control of this thesis.

4.2.1 Specific Quantizations Used

All local models were evaluated using prebuilt GGUF quantizations from Unsloth rather than quantizations produced as part of this thesis. This choice was made because the benchmark follows a deployment-constrained approach: the aim was to evaluate model configurations that could realistically be obtained and served on the available hardware, rather than to introduce an additional quantization pipeline. Using publicly available GGUF releases also improves reproducibility, since

[4]As this is an open weight model, different providers provide different quantizations of the model. Provider used FP8 quantization.

the evaluated model files correspond to concrete deployment artifacts rather than locally produced variants.

Two quantization types are used in the local model set. The small local models use `Q8_0`, an 8-bit GGUF quantization format. This was chosen for models that could fit in memory at higher precision, in order to reduce compression loss while remaining feasible for local inference. The larger models use Unsloth's `UD-Q4_K_XL` dynamic quantization. Unlike a uniform quantization scheme, Unsloth dynamic GGUFs use model-specific quantization choices across layers. The method uses a calibration dataset of more than 1.5 million tokens and evaluates quantization sensitivity using KL divergence. Layers that are more sensitive to quantization can remain at higher precision, while less sensitive layers can be compressed more aggressively [22]. This makes `UD-Q4_K_XL` suitable for larger models in this benchmark, where full 8-bit quantization would be less practical under the available VRAM constraints.

The quantization column in Table 2 should therefore be interpreted as part of the evaluated deployment configuration. The results do not isolate the effect of the base model architecture independently from quantization. Instead, each local row represents a concrete model-and-quantization pairing that was feasible to run in the local benchmark environment.

Generation-sampler parameters were not passed explicitly in the local benchmark requests or in the local `llama.cpp` launcher. Settings such as `temperature`, `top_p` and `top_k` therefore came from GGUF metadata where available. For GGUF files without sampler metadata, `llama.cpp` defaults were used. The exact values are reported in Appendix D.

4.3 ALFWorld Tasks

The benchmark uses a custom subset of the ALFWorld `valid_unseen` split. The subset contains six solvable trials, with one trial selected for each included ALFWorld task family.

The subset is deterministic rather than randomly sampled. This was necessary because candidate task folders were not equally usable: selected trials had to contain the required ALFWorld files and be marked as solvable. The resulting task suite should therefore be interpreted as a controlled benchmark subset, not as a statistically representative sample of the full ALFWorld distribution. Further details on the map subset are provided in Appendix B, specifically in Table 5.

Task type	Goal description	Scene	Expert length
Look in Light	Look at/examine a bowl with the desk lamp	`FloorPlan308`	4
Pick & Place	Put a mug in/on a desk	`FloorPlan308`	4
Clean & Place	Clean/put a clean bowl in a cabinet	`FloorPlan10`	6
Cool & Place	Cool/put a cool mug in a cabinet	`FloorPlan10`	6
Heat & Place	Heat/put a hot apple in a fridge	`FloorPlan10`	7
Pick Two & Place	Put/find two CDs in a safe	`FloorPlan308`	9

Table 3: Selected ALFWorld task subset. Expert length refers to the number of oracle actions in the expert walkthrough.

Each task is repeated 10 times per model/mode combination. This gives 60 episodes for each complete model/mode setting and keeps the task distribution identical across runs.

4.4 Benchmark Procedure

Each episode consists of one model attempting one ALFWorld task in one observation mode. At the start of an episode, the environment is reset and the initial observation is recorded. The model then repeatedly receives the current observation, selects an action through the tool-call interface described in Experimental Design, and receives the resulting ALFWorld feedback.

The benchmark uses separate system prompts for text and visual mode. The text-mode prompt describes the environment as text-based, while the visual-mode prompt states that the model receives both an image and a textual observation. Both prompts require the same action format and instruct the model to use object and receptacle names exactly as they appear in the observation. The full prompt text is provided in Appendix F.

For each turn, the model is queried through an OpenAI-compatible chat-completion interface. Requests are non-streaming, so each call returns only after the complete response has been generated. Valid action turns are executed in ALFWorld; invalid responses are logged as model turns without advancing the environment.

The ALFWorld response `Nothing happens.` is handled as an ambiguous failed-action signal. It is not treated as a simple invalid-action counter, because the same response can arise from several different situations: an incorrect model action, an unmet action precondition, an action that is valid in principle but fails in the environment, an internal environment or wrapper issue that does not crash the run or a map-specific interaction bug. The benchmark therefore does not infer a single cause from this response and does not automatically repair the action.

Instead, `Nothing happens.` is handled procedurally. The environment state is left unchanged, and the next model observation includes a short recovery message identifying the previous tool call and warning the model not to repeat the same ineffective command. This allows the episode to continue while preserving the ambiguity of the signal in the logs.

Episodes terminate after task success, reaching the maximum step limit, a model-call timeout or too many consecutive invalid tool responses. The thresholds used in all runs are 50 environment steps, 5 consecutive invalid tool responses and a 120 second client-side timeout per model call.

4.5 Measurements and Analysis

The benchmark records task outcomes, stop reasons, interaction counts, latency, token use and local GPU power measurements. These values are stored at the episode level for aggregate analysis, with turn-level records retained for inspecting individual decisions, invalid outputs and timing behavior.

4.5.1 Task Success

Task success is determined by the ALFWorld environment. An episode is counted as successful only when ALFWorld reports that the task goal has been satisfied. Success is therefore based on the executed environment state, not on the model's description of its own progress.[5]

$$\text{Success Rate} = \frac{\text{number of successful episodes}}{\text{number of attempted episodes}}$$

[5]Some models incorrectly stated that the task had been achieved before the environment reported success.

The benchmark also records the stop reason for each episode. In the final dataset, non-successful episodes stopped because of the step limit, model-call timeout, or repeated invalid tool responses.

4.5.2 Interaction Counts, Latency and Tokens

The interaction counts use the turn and step definitions established in Experimental Design. `turn_count` is used to measure the number of model attempts made during an episode, while `step_count` measures the number of valid ALFWorld actions that were actually executed. The difference between these values captures invalid tool-call behavior without treating malformed responses as environment actions.

Latency is measured as wall-clock time around each model call. Because requests are non-streaming, this measures complete response latency rather than time to first token. The analysis uses this timing information in two ways: per-action latency is computed from valid action-producing turns, while episode model-wait time is computed as the sum of model-call durations within an episode.

Token counts are taken from the usage metadata returned by the OpenAI-compatible backend. Input and output tokens are recorded separately, allowing the analysis to compare prompt/context size and generated-output volume across models and modes. If a backend omits usage metadata, token-derived summaries for that call are unavailable.

4.5.3 Local GPU Energy

GPU energy consumption is only measured for locally executable models. During each local model call, GPU power draw is sampled through NVIDIA Management Library[6]. For single-GPU models, the measurement only measures the GPU computing the inference. For dual-GPU models, the measurement uses the combined power draw of both GPUs.

Per-turn energy is computed as:

$$\text{energy}Wh = \frac{\text{average GPU power } W \times \text{wall-clock time}_{ms}}{3,600,000}.$$

Episode energy is the sum of measured turn-level energy values within the episode. The analysis computes mean and median measured GPU energy per episode for each local model/mode combination, plots the per-episode energy distribution, and reports aggregate measured GPU energy per successful episode.

This measurement captures local GPU inference energy during model calls. It does not include CPU power, RAM, storage, display power, ALFWorld or AI2-THOR overhead outside measured model calls, or energy consumed by API-hosted providers.

4.6 Methodological Scope

The benchmark is a controlled comparison of deployable model configurations on a compact ALFWorld task set. Repeating the same six tasks improves comparability across models and

[6]The Python package used was `pynvml`. The measurement is therefore based on software-reported GPU power readings.

modes, but limits task diversity. The results should therefore be interpreted as performance on this controlled subset rather than as a complete estimate of general ALFWorld performance.

The visual condition provides both a rendered image and the ALFWorld textual observation. It therefore evaluates whether adding visual input changes performance under a combined observation setting. It does not isolate image-only reasoning.

The interpretation of `Nothing happens.` responses is also limited. Since ALFWorld can return the same string for model-side mistakes, unmet pre-conditions and environment-side failures, these responses are treated as ambiguous failed-action signals rather than as direct evidence of invalid model behavior.

5 Implementation

The previous section defined the experimental design, task subset, interaction procedure and measurement definitions. This section describes how that design was implemented in the benchmark system.

5.1 Hardware setup

The local experiments used a mixed Windows and WSL setup. The Windows host ran the local llama.cpp inference server and managed GPU inference. The WSL Ubuntu environment ran the Python benchmark code, ALFWorld, AI2-THOR visual mode, result writing and analysis. The benchmark process communicated with the local server through the OpenAI-compatible `llama-server` endpoint on localhost.

This split was mainly practical. The local GGUF models were served through a Windows llama.cpp build, while the ALFWorld benchmark environment was run from Linux.[7] Running llama.cpp on Windows rather than inside WSL also proved to be the more stable and performant choice in practice, particularly due to the NVIDIA GPUs. In text mode, WSL only had to run the ALFWorld text environment and send model requests to the server. In visual mode, WSL also had to run AI2-THOR/Unity through the WSL graphical stack.

The local workstation used an AMD Ryzen 7 5700X CPU, 32 GB of system memory and two NVIDIA GeForce RTX 5060 Ti GPUs with 16 GB of VRAM each. Smaller local models were served on a single GPU, while larger quantized models were split across both GPUs. This made it possible to run models that would not fit comfortably on one 16 GB card, but it did not make the setup equivalent to a single GPU with 32 GB of unified VRAM.

The hardware setup is part of the experimental condition. The reported latency, timeout behavior and measured local GPU energy depend not only on the model files, but also on quantization, the `llama.cpp` serving configuration, GPU split mode, PCIe layout and the Windows/WSL runtime environment. Different hardware, especially a single GPU with larger VRAM and higher memory bandwidth, could produce substantially different local latency and energy results. More detailed hardware limitations and the dual-GPU split choice are discussed in Appendix A.

[7] ALFWorld does not support Windows.

Before the final local runs, the WSL configuration was adjusted to improve memory headroom for long visual-mode runs. The exact settings and motivation are described in subsection C.1.

Component	Configuration
CPU	AMD Ryzen 7 5700X
RAM	32 GB
GPUs	2 × NVIDIA GeForce RTX 5060 Ti (16 GB)
Local model serving	Windows llama.cpp server
Benchmark environment	WSL Ubuntu
WSL allocation	20 GB memory, 16 GB swap

Table 4: Local hardware and runtime configuration used for the local benchmark runs.

5.2 Software setup

The benchmark was implemented as a Python project around a repository-local ALFWorld setup. The runtime environment used a Conda-based Python environment with ALFWorld installed in editable mode. Important packages included ALFWorld, AI2-THOR, OpenAI, NumPy, and pynvml.

The benchmark code had four main responsibilities. First, it prepared the selected ALFWorld subset and patched the ALFWorld configuration. Second, it ran benchmark episodes by repeatedly sending observations to a model and executing the returned tool call in ALFWorld. Third, it handled local model serving by coordinating the Windows llama.cpp server from WSL. Fourth, it loaded the finalized result files and generated the analysis outputs used in the Results section.

The ALFWorld setup used a local editable installation rather than a global installation to allow modifications to ALFWorld. The setup script copied the selected six-task subset into a custom directory, patched the ALFWorld configuration to evaluate that subset and set the visual resolution to 1280×720 pixels[8] In addition, two local compatibility patches were applied to the ALFWorld code: one added a missing receptacle-state field used by the visual controller and the other corrected an oracle instance-segmentation issue. These changes were required to make the local ALFWorld environment run reliably enough for the benchmark. More information about this can be found in Appendix B.

5.3 Inference Backends

All models were accessed through the same OpenAI-compatible provider wrapper. This wrapper stored the per-episode conversation history, sent non-streaming chat-completion requests, recorded timing and token metadata, and attached the current AI2-THOR frame in visual mode. This common interface allowed local and API-hosted models to be evaluated with the same benchmark loop.

Local inference was served by llama.cpp on Windows. The llama.cpp was built from source[9]

[8]The original resolution for the visual environment was 300×300 pixels. It was increased to 720p to more resemble resolution and aspect ratio that is used for commercial cameras.

[9]As opposed to using a pre-built binary.

using llama.cpp commit `e95dae18d64ae4471d61a9dc87880a64e0e5c86e`. The local server was configured with a 65 536-token context size, all layers offloaded to GPU(s), flash attention and F16 KV-cache. The server-side token generation cap per response was set to 8192 tokens. The benchmark also enforced a 120 second client-side timeout per model call (immediately ending the run).

The final local workflow used the `llama-server` in router mode. Instead of manually starting a separate server for every model, the router was started once with the configured local models available, but unloaded. The WSL-side orchestration script then loaded one model, ran the benchmark for that model, unloaded it and moved to the next model.[10] This way only one model was kept in memory and it made the long runs manageable.

Two llama.cpp settings were added for stability for all models after earlier Out-Of-Memory failures with Gemma-4-31B: the in-RAM prompt cache was disabled and context checkpoints were limited to 1. More details can be found in subsection C.2.

The API-hosted models were served through OpenRouter using the same provider wrapper used for the local models. These runs used the same benchmark loop as the local models, but did not use the local `llama-server`.

5.4 Execution, Results, Reproducibility

The benchmark runner saved one JSON file for every completed episode. Each file stored episode-level metadata, the turn-level interaction log and aggregate summary values. This made it possible to inspect both the final task outcome and the intermediate model-environment interactions, including tool calls, timing information, token counts and GPU power measurements. An example of how a JSON file is structured can be found in Appendix E.

The final analysis dataset was selected explicitly. Earlier diagnostic, testing, partial and failed runs were excluded. Result files produced by earlier code versions or different llama.cpp builds were also excluded, even when the files themselves were valid JSON outputs.

The local results should be interpreted as results for the full local deployment, not only for the named model families. Reproducing them would require using the same ALFWorld subset, local ALFWorld patches, benchmark configuration, llama.cpp build, GGUF model files, multimodal projector files, GPU loading mode and Windows/WSL setup. These details can affect runtime behavior, especially latency, memory use and visual-mode stability. For API-hosted models, the benchmark prompts and request format can be reproduced, but the provider-side hardware and serving configuration remain outside local control.

6 Code

The source code for the benchmark implementation and analysis is available at:

https://github.com/aflokk/alfworld-llm-benchmark

[10] Initially with this approach, memory leak: the llama-server not freeing up memory properly due to repeated model loading and unloading was a concern, however this was not actually observed during benchmarking.

7 Results

The final dataset contains 1 020 completed benchmark episodes across 17 model/mode groups, with 816 successful episodes and an overall success rate of 80.0%. The benchmark runs recorded 42.08 hours of total runtime including timeouts and cleanup, and processed 162.7 million tokens in total: 157.8 million input tokens and 4.91 million output tokens. The local GPUs consumed 5.12 kWh of energy during model calls.

7.1 Success Rate by Model and Mode

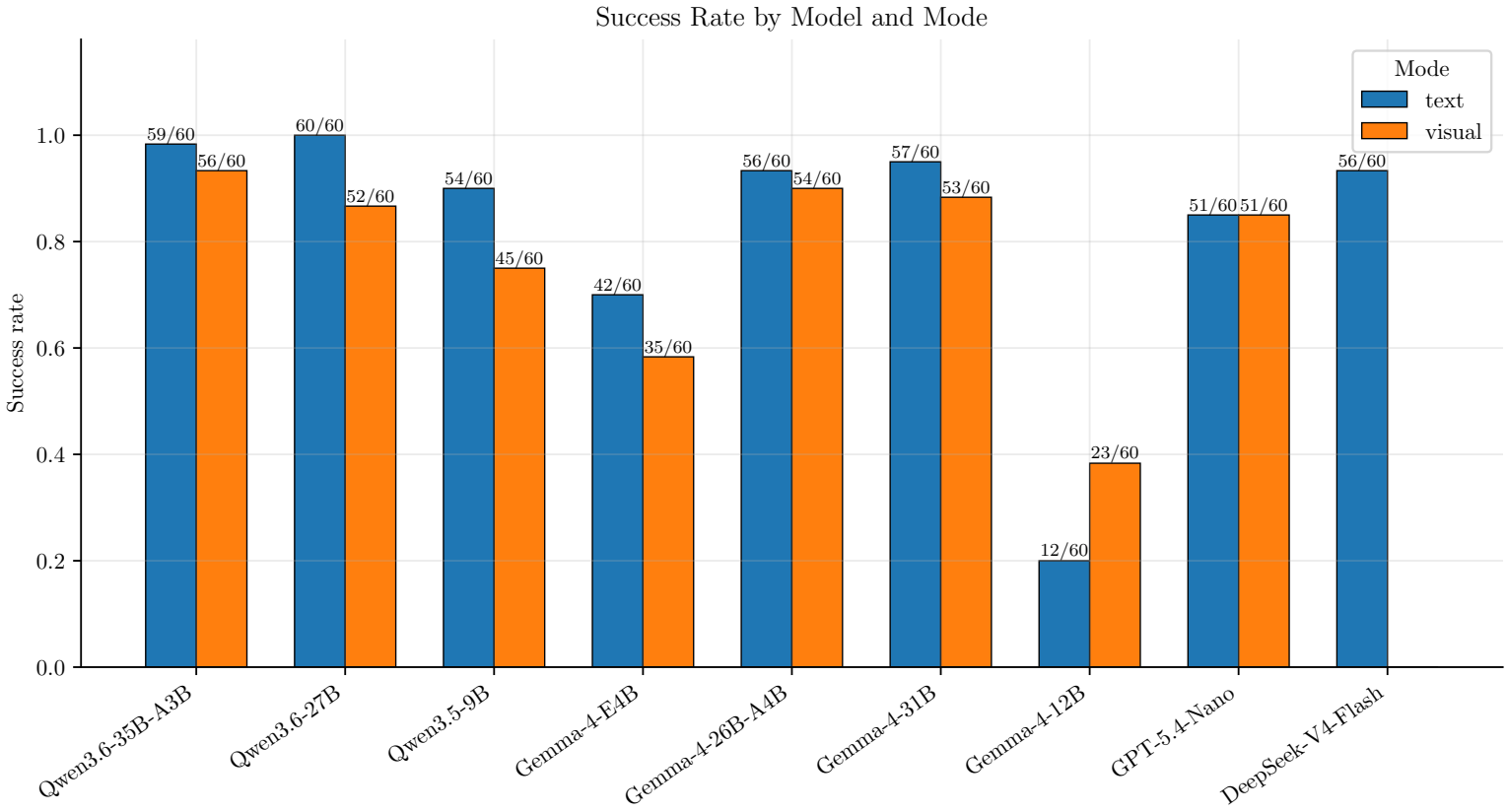

Figure 1: Episode success rate by model and input mode. Each model/mode group contains 60 episodes; bar labels report successful episodes out of 60.

Figure 1 shows substantial variation between models. The best individual result is Qwen3.6-27B in text mode, which succeeds in all 60 episodes. Qwen3.6-35B-A3B is nearly perfect in text mode with 98.3% (59/60) successes and remains the strongest local visual model with 93.3% (56/60) successes. Strong local results are also obtained by Gemma-4-31B in text mode (95.0%, 57/60), Gemma-4-26B-A4B in text mode (93.3%, 56/60), and Gemma-4-26B-A4B in visual mode (90.0%, 54/60).

The lowest-success model is Gemma-4-12B with 20.0% (12/60) successes in text mode and 38.3% (23/60) in visual mode. By success rate alone, this separates it from the rest of the model set.

Excluding this outlier, the weakest model with results in both modes is Gemma-4-E4B, with 70.0% (42/60) successes in text mode and 58.3% (35/60) in visual mode.

Across the complete dataset, text mode has a higher observed success rate than visual mode: Text mode succeeds in 82.8% (447/540) episodes, while visual mode succeeds in 76.9% (369/480) episodes. The same direction is also visible within the local model subset, where every model has both text and visual runs: local text mode succeeds in 81.0% (340/420) episodes, compared with 75.7% (318/420) episodes for local visual mode. The API-hosted results should be read more separately. GPT-5.4-Nano is stable across modes, with 85.0% (51/60) successes in both text and visual mode, while DeepSeek-V4-Flash reaches 93.3% (56/60) in text mode but has no visual data (due to not supporting image inputs).

Overall, these results do not show a systematic success-rate advantage from adding visual input in this benchmark setup. For most models with both text and visual runs, the visual condition is equal to or below the corresponding text condition. The exception is Gemma-4-12B, where visual mode improves from 20.0% (12/60) to 38.3% (23/60), but this improvement is from a very low text baseline and does not make the model competitive with the stronger groups.

7.2 Task-Level Success

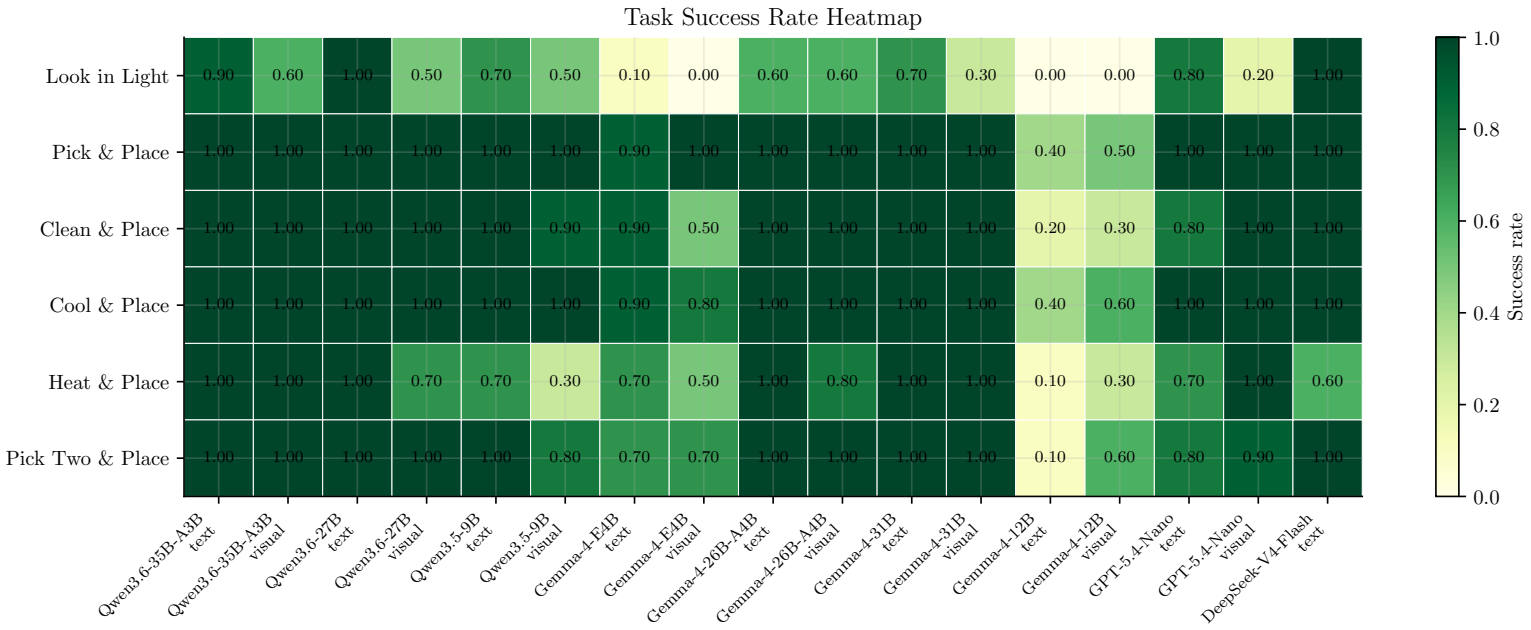

Figure 2: Task-level success rate by model, mode, and ALFWorld task family. Each heatmap cell is based on 10 episodes from one repeated task family.

Figure 2 breaks the success rates down by task family. Each model/mode group repeats the six-task subset 10 times, so each heatmap cell corresponds to 10 episodes. This means each cell is based on a small number of runs and should be interpreted as an approximate task-level comparison rather than a precise statistical estimate.

The easiest task families are Pick & Place and Cool & Place. Pick & Place succeeds in 92.9% (158/170) episodes and Cool & Place succeeds in 92.4% (157/170) episodes. These tasks are almost solved by all stronger models. The remaining failures are concentrated mainly in the lower-performing groups, especially Gemma-4-12B, with smaller drops in Gemma-4-E4B and other timeout-affected runs.

Clean & Place and Two Objects (Pick Two & Place) form a second tier, both at 85.9% (146/170) success. Two Objects has the longest expert walkthrough in the selected subset (9), but it is still solved reliably by most strong models. This suggests that the difficulty pattern is not explained by the expert trajectory length alone.

The main bottleneck is Look in Light. It succeeds in only 50.0% (85/170) episodes, and the visual version is especially weak: visual Look in Light succeeds in 33.8% (27/80) episodes, compared with 64.4% (58/90) episodes in text mode. Several otherwise strong visual models lose most of their failures in this task family. For example, GPT-5.4-Nano visual succeeds in only 20.0% (2/10) Look in Light episodes, while Gemma-4-31B visual succeeds in 30.0% (3/10) episodes.

Heat & Place is also more difficult than the basic Pick & Place variants, with 72.9% (124/170) success. It separates models that otherwise perform similarly on simpler tasks. For instance, Qwen3.5-9B visual drops to 30.0% (3/10) success on Heat & Place, while stronger groups such as Qwen3.6-35B-A3B text and visual solve all ten Heat & Place episodes.

7.3 Stop Reasons

Figure 3 shows why non-success episodes terminated. Across the full dataset, 10.8% (110/1,020) of episodes stop because they reach the maximum step count, 9.1% (93/1,020) stop because of a timeout, and 0.1% (1/1,020) stop because of too many invalid tool responses. Episode-ending tool-call-format failures were therefore rare in the final dataset. The main recurring failure mode is reaching the step limit, while timeouts are concentrated in a smaller number of model/mode groups.

The failure mix differs by model. Most high-performing groups fail mainly by max-step termination. This stop reason only indicates that the episode reached the configured step limit without satisfying the ALFWorld success condition. It does not distinguish between different underlying behaviours, such as ineffective exploration, repeated action patterns, incorrect object interactions, or partial progress that was not completed in time. This pattern appears in models such as Qwen3.6-35B-A3B visual, Qwen3.5-9B visual, Gemma-4-E4B visual, GPT-5.4-Nano, and DeepSeek-V4-Flash.

Gemma-4-12B has a distinct stop-reason profile. In text mode, Gemma-4-12B has 47/60 timeout episodes; in visual mode, it has 37/60 timeout episodes. This separates it from the other lower-performing groups, where failure is more often caused by reaching the step limit. For example, Gemma-4-E4B visual has 25/60 max-step failures, Gemma-4-E4B text has 17/60 max-step failures, and Qwen3.5-9B visual has 15/60 max-step failures. A smaller number of timeouts also occurs in Qwen3.6-27B visual and Gemma-4-31B visual, with 4/60 timeout episodes each. The cloud/API-hosted groups have no timeouts in this dataset; their failures are all max-step failures.

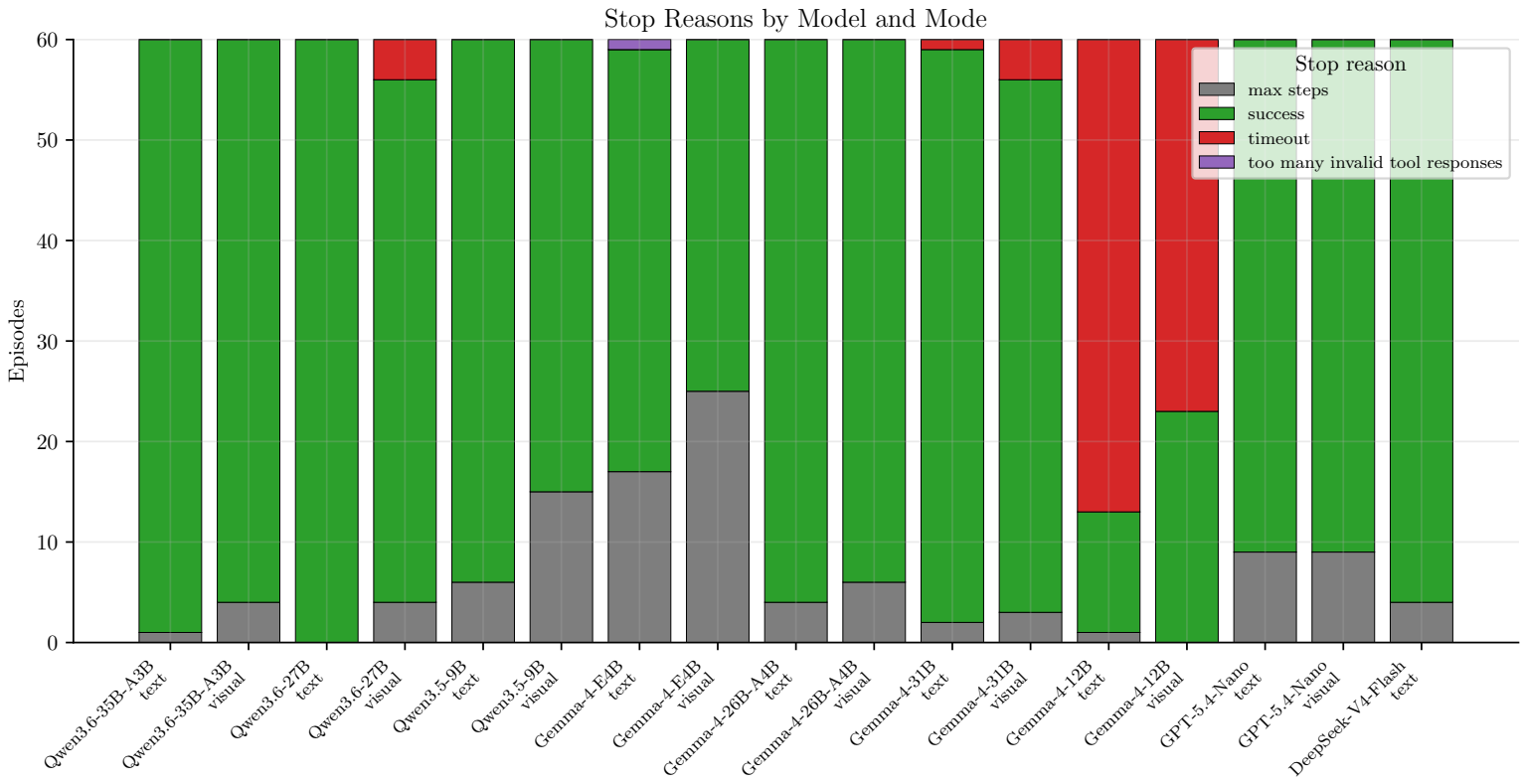

Figure 3: Episode stop reasons by model and mode. Each stacked bar contains 60 episodes and separates successful episodes from failures caused by the step limit, model-call timeout, or repeated invalid tool responses.

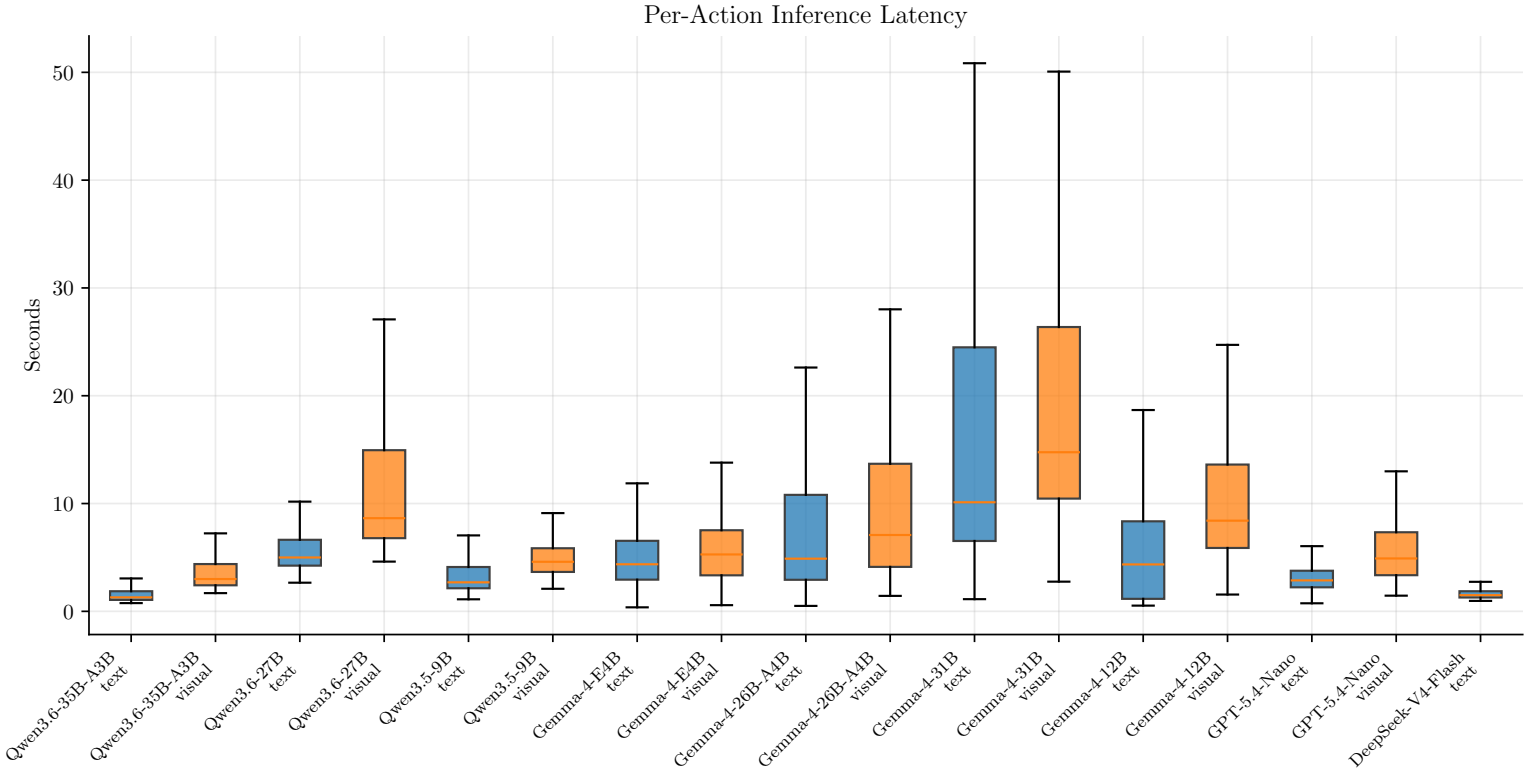

Figure 4: Per-action model-call latency for valid executed actions. The distribution is computed over turns that produced a valid tool call and were executed as ALFWorld actions.

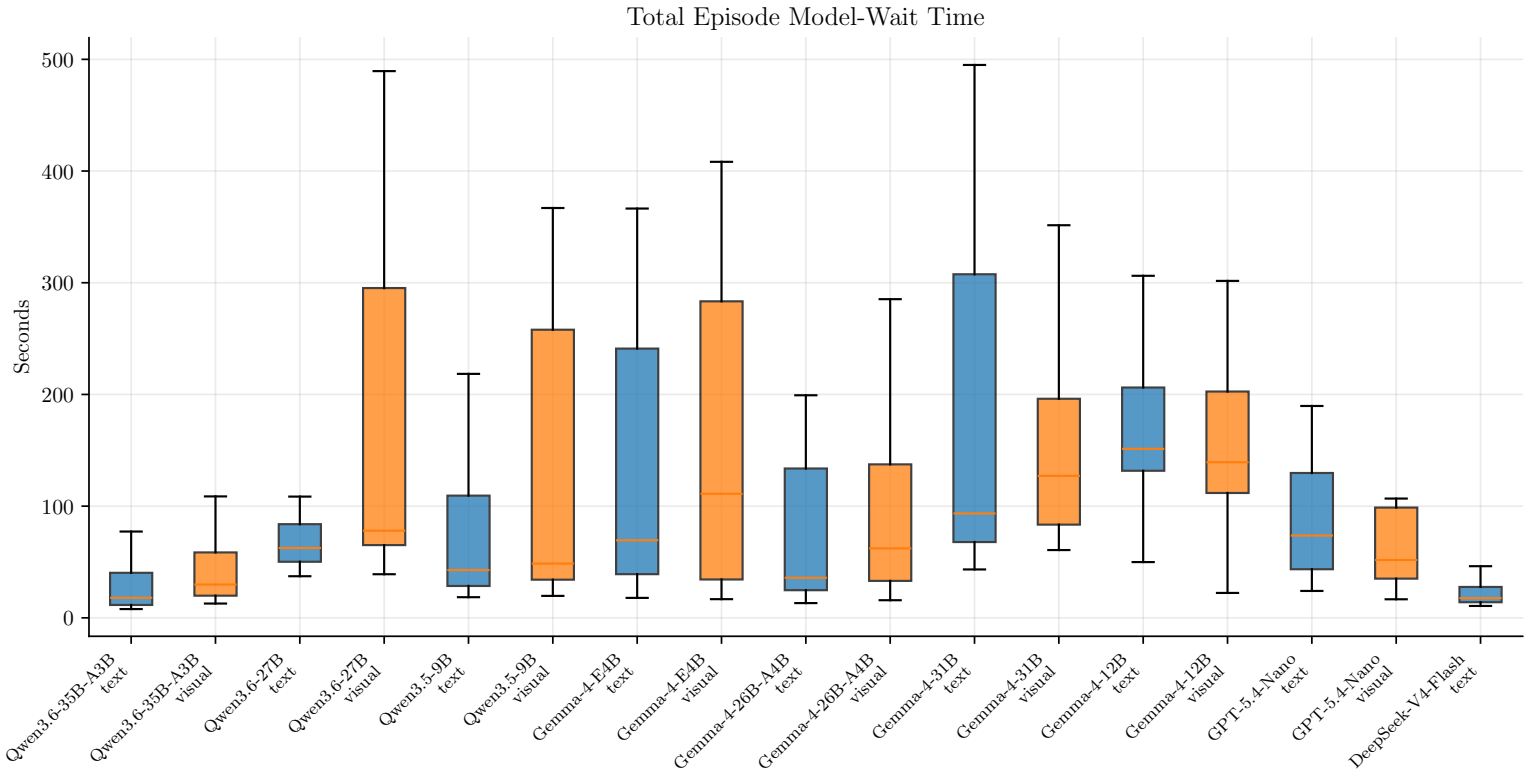

Figure 5: Total episode model-wait time by model and mode. The value is the sum of model-call waiting time within an episode and does not include ALFWorld environment execution time or local backend cleanup waits after timeout.

7.4 Latency

Figure 4 reports latency per valid executed action. The fastest local median action latency is Qwen3.6-35B-A3B text at 1.29 seconds. DeepSeek-V4-Flash text is the fastest API-hosted datapoint, almost matching Qwen3.6-35B-A3B text. The fastest visual model in this dataset is also Qwen3.6-35B-A3B, with a median action latency of 3.00 seconds.

Visual mode increases median action latency for every model that has both text and visual runs. This is expected, as visual mode sends image input in addition to the text observation, requiring more computation. However, the size of the increase differs strongly by model. Qwen3.6-35B-A3B remains relatively fast in visual mode, while Gemma-4-31B visual has the slowest median action latency at 14.76 seconds.

Figure 5 reports episode model-wait time, which sums model-call waiting time across the whole episode. This metric is important because per-action latency alone does not capture how many actions a model uses. A model can have moderate latency per action but still accumulate a high episode time if it takes many actions, loops, reaches the maximum step count or times out.

DeepSeek-V4-Flash text and Qwen3.6-35B-A3B text have the lowest median episode model-wait times, both around 18 seconds. Qwen3.6-35B-A3B visual also has a low median episode wait for a visual model at 29.83 seconds. In contrast, Gemma-4-31B has high mean episode wait in both modes, close to 5 *minutes* on average. Several groups have means much larger than medians, especially Qwen3.6-27B visual and Gemma-4-31B.

7.5 Measured Local GPU Energy

Figure 6 reports the distribution of measured local GPU energy per episode, while Figure 7 reports total measured local GPU energy divided by the number of successful episodes in each model/mode group. These two views answer different questions. Per-episode energy describes the typical measured GPU cost of running an episode, regardless of whether the task succeeds. Energy per successful episode combines energy use with task completion and therefore increases when a model either consumes more energy or succeeds in fewer episodes.

Qwen3.6-35B-A3B has the most favorable local energy profile among the high-success models. Its median episode energy is 0.74 Wh in text mode and 1.04 Wh in visual mode. After normalizing by successful episodes, it remains the most energy-efficient local model/mode group, with 1.38 Wh per success in text mode and 2.28 Wh per success in visual mode.

This normalization changes the interpretation of the energy results. Qwen3.6-27B is the only model/mode group with perfect success, but it requires 4.97 Wh per successful episode, more than three times the Qwen3.6-35B-A3B text value. Gemma-4-31B also reaches high success, especially in text mode, but it has the highest energy per successful episode in both modes: 16.28 Wh in text mode and 16.74 Wh in visual mode.

Gemma-4-12B shows how low success can affect the energy-per-success metric. Its median episode energy is not the highest, but only 20.0% (12/60) of its text-mode episodes succeed. The resulting energy per successful episode is therefore high, at 15.36 Wh.

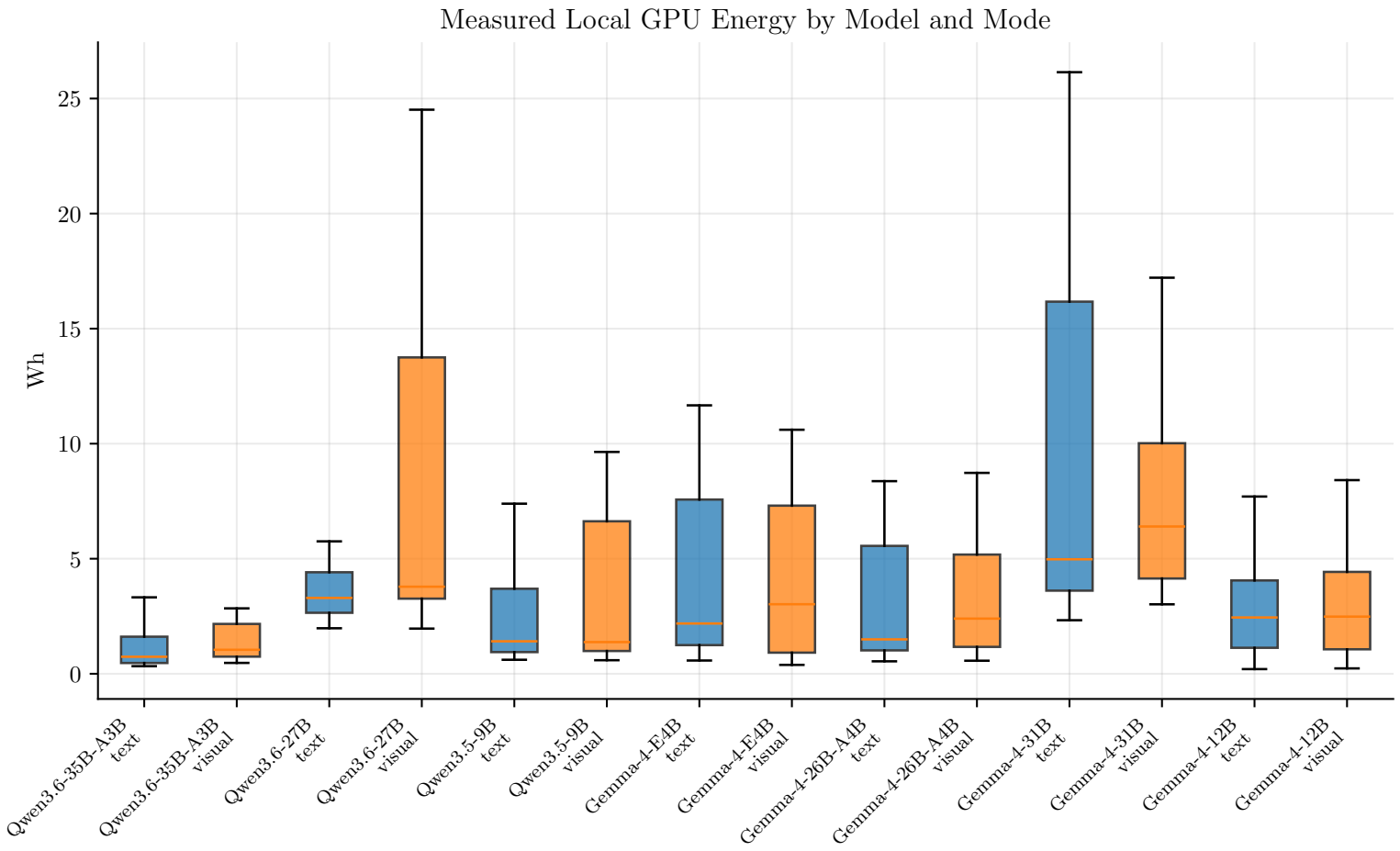

Figure 6: Measured local GPU energy per episode for local-provider models. Energy is computed from measured GPU power during model calls and summed over each episode.

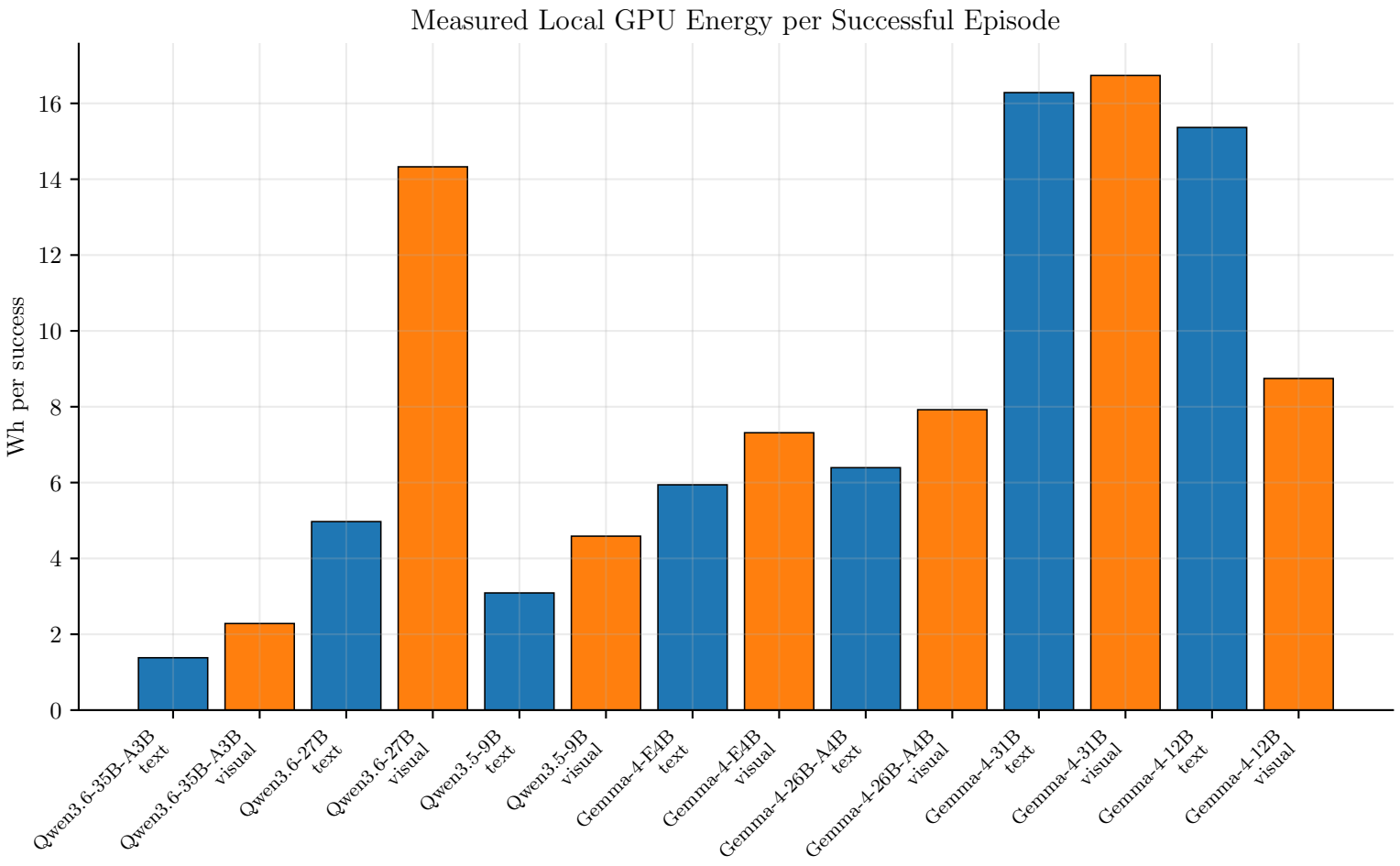

Figure 7: Measured local GPU energy per successful episode for local-provider models. The value is total measured GPU energy for a model/mode group divided by the number of successful episodes in that group.

7.6 Success, Latency, and Energy Trade-off

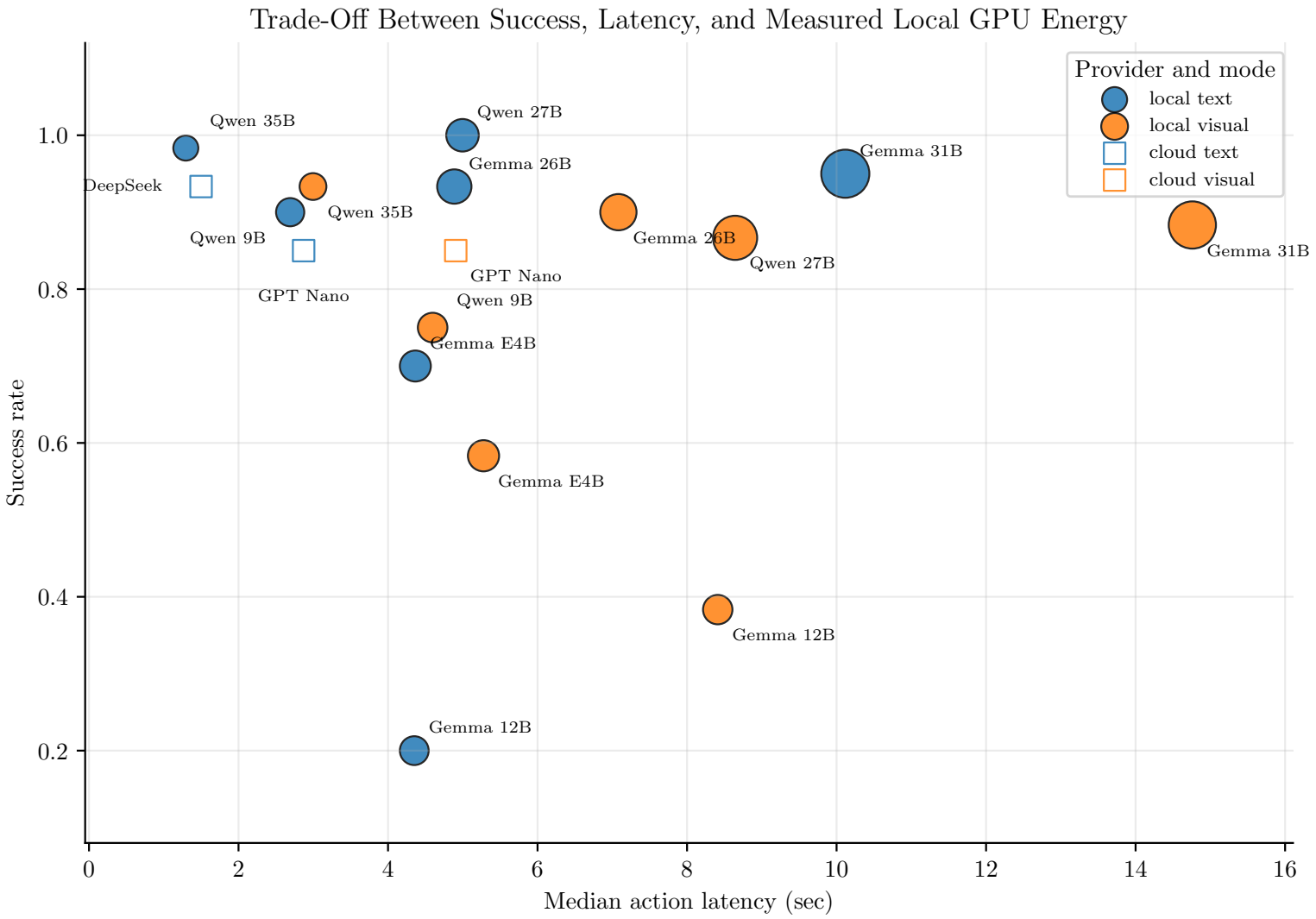

Figure 8: Trade-off between success rate, median action latency, and measured local GPU energy. The x-axis shows median per-action latency, the y-axis shows episode success rate, and marker area represents mean measured local GPU energy per episode for local-provider models. Hollow markers denote API-hosted models, for which local GPU energy was not measured.

Figure 8 summarizes the main trade-off. The strongest local trade-off model is Qwen3.6-35B-A3B in text mode: it reaches 98.3% (59/60) success, has a median action latency of 1.29 seconds and has the lowest measured local energy per successful episode. Qwen3.6-27B text reaches the highest possible success rate, but it is slower and more energy intensive.

Among visual models Qwen3.6-35B-A3B again gives the strongest balance, with 93.3% (56/60) success and a median action latency of 3.00 seconds. Gemma-4-26B-A4B visual is also strong in success rate at 90.0% (54/60), but it has higher latency and higher measured energy. Gemma-4-31B reaches high success, but its latency and energy place it in a less favorable region of the trade-off space.

The API-hosted models are competitive on success and latency. DeepSeek-V4-Flash text reaches 93.3% (56/60) success with a median action latency of 1.50 seconds, making it a strong cloud/API-hosted text result. GPT-5.4-Nano is stable across text and visual mode at 85.0% (51/60) success in both conditions, with moderate latency.

Overall, the results show that the highest-success model is not necessarily the best efficiency trade-off. Qwen3.6-27B text achieves perfect success, but Qwen3.6-35B-A3B text achieves nearly the same success with much lower latency and measured local GPU energy. The results also show that adding visual input does not improve success overall in this benchmark setup, while it consistently increases action latency for models with both text and visual runs.

8 Discussion

8.1 Answering the Research Questions

The results show that lightweight local models can be capable high-level task planners in the evaluated ALFWorld setting, but that this capability depends strongly on the model architecture, serving configuration and deployment constraints. For **SQ1**, the local models were not categorically worse than the API-hosted models. Several local model/mode groups reached success rates close to or above the API-hosted comparison points. In particular, `Qwen3.6-27B` text solved all episodes, while `Qwen3.6-35B-A3B` text reached nearly the same success rate with substantially lower latency and measured local GPU energy. This suggests that replacing a cloud-based model with a local model does not necessarily imply a large loss in task success, **at least for this controlled ALFWorld subset.**

For **SQ2** and **SQ3**, however the differences between local models are large. The results do not support a simple "larger is better" interpretation. `Gemma-4-31B` reached high task success, but had high latency and energy use. In contrast, `Qwen3.6-35B-A3B` achieved a stronger overall balance between success, latency and measured energy. This is most likely due to the advantage of Mixture-of-Experts Models (MoE) under the tested hardware. These models could use a larger total parameter budget while activating fewer parameters per token, which made them better suited to the limited VRAM bandwidth of the dual consumer GPU setup, than similarly large dense models.

For **SQ4**, the most important trade-off is therefore not simply cloud versus local, but model capability relative to inference cost. The best local trade-off model in this benchmark was `Qwen3.6-35B-A3B`. The API/Cloud models remained competitive, especially `DeepSeek-V4-Flash` in text mode, but they also introduce the deployment considerations discussed in Deployment Settings. For this benchmark `GPT-5.4-Nano` cost 2.35$, while `DeepSeek-V4-Flash` cost 0.256$ (89% cheaper). While this exact ratio will shift as API prices change, it illustrates how much cloud-model selection affects deployment costs.

8.2 Visual Mode

One of the clearest results is that adding visual frames did not improve task success in this benchmark setup. As shown in Figure 1, text mode outperformed visual mode overall and visual input also increased per-action latency. This result should not be interpreted as evidence that

visual information is unimportant for robotics. Instead, it reflects the specific observation structure used here: visual mode gives both the rendered frame and the text observation.

The text observation was the more reliable state representation. It described relevant objects, receptacles and object IDs directly, while the rendered frame only showed the current camera view. The frame did not always contain all objects mentioned in the text, and its usefulness depended on view angle, distance and framing. This limitation is illustrated in Figure 9, where the target object is present in the scene but is difficult to identify from the rendered frame before pickup. Under these conditions, the image usually added computation without adding useful decision information. For robotics settings incorporating these smaller LLMs/VLMs a more modular architecture may be preferable. A perception component can detect objects and produce a structured state representation, while the language model receives that structured representation for planning. In other words, these results support the use of LLMs/VLMs as high-level planners inside a larger modular system, rather than a drop-in replacement for perception, state tracking and control.

8.3 Task Difficulty

The results indicate that several task types were handled reliably by the stronger models. Pick & Place, Cool & Place, Clean & Place and Pick Two & Place tasks were solved at high rates, even though they required multiple ordered actions. Compared to the earlier reseach which also evaluated on ALFWorld (mentioned in Related Work), specifically ReAct[24] and AgentBench[13] these results are noticeably stronger. The scores are not directly comparable, however general success rate trends suggest that LLMs have improved substantially on ALFWorld-style high-level planning.

The main exception was the Look in Light task. As shown in Figure 2, this task was the largest task-level bottleneck especially in visual mode. This was not simply because the task required long-horizon planning. Instead, the task exposed a mismatch between the natural-language instruction and the ALFWorld success condition. The instruction suggests agent should look at or examine a bowl under a desklamp. The actual environment goal is narrower: the agent must be at the receptacle where the lamp is located, hold the bowl and toggle the lamp. No explicit `look` or `examine` action is required for success.

This mismatch explains many of the observed failures. Actions such as examining the bowl, examining the lamp, moving the bowl to the receptacle with the lamp or trying to take the desklamp are reasonable responses to the natural language instruction, but they do not satisfy the hidden ALFWorld goal requirements. This means that many Look in Light failures should not be interpreted as pure planning failures, but rather as failures caused by the benchmark interface, where the models pursued the task as it was worded, while the environment required a narrower step sequence.

8.4 Failure Modes

The stop-reason results in Figure 3 show that most non-successes were caused by max-step termination or timeout. Max-step termination is the expected failure mode for an agent that

(a) Before pickup. The agent is positioned near the sink, but the apple is only barely visible in the rendered frame.

(b) After pickup. The next frame shows the same interaction context after the agent has taken the apple from the sink.

Figure 9: Example of a visual observation where the rendered frame provides limited additional information. Although the apple was available to the agent, it was difficult to identify visually before pickup because of the view angle, distance and framing. This illustrates why the image input was not always a reliable substitute for the structured text observation.

remains syntactically valid but fails to complete the task. It usually means that the model kept attempting actions, searched inefficiently, repeated ineffective patterns, or made partial progress without satisfying the final ALFWorld success condition.

The structured tool-call interface appears to have worked reliably. Only one episode ended because of too many invalid tool responses. This is a positive result, because incorrect output formatting had been an issue during early development especially for smaller models such as `Gemma-4-E4B`. The final tool-call setup therefore seems to have addressed syntax failures.

Timeouts require a different interpretation. A timeout can mean that a model was responding too slowly, but it can also mean that inference entered an infinite-loop. The result data alone cannot separate these cases. For the largest dense models, such as `Qwen3.6-27B` and `Gemma-4-31B`, occasional timeouts are plausible consequences of slow generation under the local serving setup. `Gemma-4-12B` is different. It produced far more timeouts than any other model in both text and visual mode. During testing, this model also frequently failed to stop generation and continued until the context limit was reached[11]. Therefore, its results should be treated as an outlier for this specific deployment configuration. They do not support a strong conclusion about the underlying model family in general, but they do show that this particular quantized model, chat template and llama.cpp serving configuration was not reliable for the benchmark.

8.5 Deployment Implications

The three deployment settings from Deployment Settings lead to different model choices.

For Cloud-Based Inference, the main decision is the balance between task success, latency, privacy requirements, provider trust and operating cost. Cloud models are easy to replace when newer models are available and they avoid local hardware maintenance. However, they also create dependence on provider availability, pricing, model updates and data-handling policies.

For Local-Server Inference, the strongest candidate in these results is `Qwen3.6-35B-A3B`. It combined high success, low action latency and low measured energy per successful episode. This makes it more attractive than a slower dense model that reaches similar success but requires more time and energy. The result also shows why energy should not be interpreted separately from success: a model that uses little energy per episode can still be inefficient per completed task if it fails often.

For On-Device Inference, the constraints are stricter. A small model such as `Gemma-4-E4B` is interesting because it has a much smaller memory footprint and still had a 64.2% success rate. However, compared to the other models its lower success rate means it would need additional system-level support before being suitable for a reliable robotic system. Such support could include better state tracking, explicit recovery policies, task-specific validators or a fallback to a larger model when confidence is low. Whether the measured power draw would be feasible for an actual mobile robot is outside the scope of this thesis.

Overall, these results support the view that current small and medium open-weight models can be useful high-level planning components, but not standalone "robot brains". The most practical architecture is likely a system in which perception, state estimation, affordance and low-level control

[11] This is why the timeouts and the max token generation limits had to be introduced.

are handled by separate components, while the model is responsible for selecting high-level actions from a well defined action space.

9 Conclusion

This thesis evaluated whether locally run open-weight language and multimodal models can provide a practical alternative to cloud-based models for high-level embodied task planning. Using a controlled ALFWorld benchmark with text and visual observation modes, the results show that local inference can be competitive, but that the outcome depends strongly on the specific model and serving setup. The best local trade-off was achieved by `Qwen3.6-35B-A3B`, which combined high task success with relatively low latency and low measured local GPU energy.

The results also show that visual input did not automatically improve task performance in this benchmark. Text-only observations were often sufficient, while visual mode increased latency and introduced additional dependence on camera framing and environment state visibility. Overall, the findings suggest that local models are a viable option for controlled embodied planning tasks, but not as standalone robotic systems. They are most useful when combined with structured state tracking, reliable perception, validation mechanisms and recovery behavior. Local deployment should therefore be evaluated as a system-level trade-off between success rate, latency, energy use, reliability and deployment control, rather than as a direct replacement for cloud inference alone.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020.

[2] Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Ruo Yu Tao, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. Textworld: A learning environment for text-based games. In *Computer Games: 7th Workshop, CGW 2018*, volume 1017 of *Communications in Computer and Information Science*, pages 41–75. Springer, 2019.

[3] Jiafei Duan, Samson Yu, Hui Li Tan, Hongyuan Zhu, and Cheston Tan. A survey of embodied ai: From simulators to research tasks. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 6(2):230–244, 2022.

[4] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[5] Georgi Gerganov and contributors. llama.cpp: Llm inference in C/C++. https://github.com/ggml-org/llama.cpp, 2026. Version used: commit e95dae18d64ae4471d61a9dc87880a64e0e5c86e; accessed 2026-07-01.

[6] Georgi Gerganov and contributors. llama.cpp quantization tool documentation. https://github.com/ggml-org/llama.cpp/blob/master/tools/quantize/README.md, 2026. Accessed: 2026-07-01.

[7] GGML Org. Using multiple GPUs with llama.cpp. https://github.com/ggml-org/llama.cpp/blob/master/docs/multi-gpu.md, 2026. Accessed: 2026-07-01.

[8] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 9118–9147. PMLR, 2022.

[9] Erik Johannes Husom, Arda Goknil, Merve Astekin, Lwin Khin Shar, Andre Kåsen, Sagar Sen, Benedikt Andreas Mithassel, and Ahmet Soylu. Sustainable llm inference for edge ai: Evaluating quantized llms for energy efficiency, output accuracy, and inference latency, 2025.

[10] International Energy Agency. Energy and ai. Technical report, IEA, Paris, 2025. Licence: CC BY 4.0; accessed 2026-07-01.

[11] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

[12] Chia Xin Liang, Pu Tian, Caitlyn Heqi Yin, Yao Yua, Wei An-Hou, Li Ming, Xinyuan Song, Tianyang Wang, Ziqian Bi, and Ming Liu. A comprehensive survey and guide to multimodal large language models in vision-language tasks, 2024.

[13] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

[14] Yang Liu, Weixing Chen, Yongjie Bai, Xiaodan Liang, Guanbin Li, Wen Gao, and Liang Lin. Aligning cyber space with physical world: A comprehensive survey on embodied ai, 2024.

[15] Sasha Luccioni, Yacine Jernite, and Emma Strubell. Power hungry processing: Watts driving the cost of ai deployment? In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '24, pages 85–99. ACM, 2024.

[16] NVIDIA. Compare geforce graphics cards. https://www.nvidia.com/en-us/geforce/graphics-cards/compare/#compare-50, 2026. Accessed: 2026-07-01.

[17] Mohaimenul Azam Khan Raiaan, Md. Saddam Hossain Mukta, Kaniz Fatema, Nur Mohammad Fahad, Sadman Sakib, Most Marufatul Jannat Mim, Jubaer Ahmad, Mohammed Eunus Ali, and Sami Azam. A review on large language models: Architectures, applications, taxonomies, open issues and challenges. *IEEE Access*, 12:26839–26874, 2024.

[18] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. From words to watts: Benchmarking the energy costs of large language model inference. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2023.

[19] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[20] Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10740–10749, 2020.

[21] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[22] Unsloth. Unsloth Dynamic 2.0 GGUFs. https://unsloth.ai/docs/basics/unsloth-dynamic-2.0-ggufs, 2026. Accessed: 2026-07-01.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[24] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[25] Liangqi Yuan, Dong-Jun Han, Shiqiang Wang, and Christopher G. Brinton. Local-cloud inference offloading for llms in multi-modal, multi-task, multi-dialogue settings, 2025.

A Extended Notes on Hardware

The local hardware setup was chosen because it was recent consumer hardware that was available for this project. For reproducibility NVIDIA driver version 595.97 was used with CUDA 13.2.

This setup had practical advantages. Two RTX 5060 Ti cards are substantially cheaper to obtain than a single RTX 5090, while still providing a total of 32 GB of installed VRAM across the two cards. The GPUs also had much lower (combined) power requirements than a high-end card such as the RTX 5090. NVIDIA lists the RTX 5060 Ti with a 180 W total graphics power, while the RTX

5090 is listed with a 575 W total graphics power [16]. This made the used setup more attainable and easier to use.

The main limitation was that the two GPUs did not provide one unified 32 GB memory pool. Each card had its own 16 GB of VRAM. For the larger local models, the selected quantized model files exceeded what could be loaded on a single 16 GB card, even before accounting for the memory required by the context window. Those models therefore had to be split across both GPUs. This allowed the benchmark to run them locally, but it was not equivalent to running the same models on a single GPU with enough VRAM.

Memory bandwidth was another important limitation. The RTX 5060 Ti uses a 128-bit memory interface, while the RTX 5090 uses a 512-bit memory interface (both on GDDR7). This results in effective memory bandwidth of 448 GB/s for the RTX 5060 Ti 16 GB compared to 1792 GB/s for the RTX 5090 (4× difference)[16]. This difference matters for local LLM inference because token generation can be limited by memory movement rather than raw compute alone. During the local runs, GPU monitoring also suggested that the cards were often not fully compute-saturated, which is consistent with memory bandwidth and data movement being important bottlenecks.

A.1 Dual-GPU Splitting Choice

The larger models which could not be loaded on a single 16 GB GPU were **evenly** split across both cards:

```
CUDA_VISIBLE_DEVICES=0,1
--split-mode layer
--tensor-split 1,1
```

In this configuration, `llama.cpp` uses layer splitting. Whole model layers are distributed across the two GPUs. The `--tensor-split 1,1` argument sets the split ratio between the two cards, so the model is distributed evenly. This increases the amount of model data that can be kept in GPU memory, but it does not make the two cards behave like one unified 32 GB GPU. [7]

The alternative `--split-mode tensor` was also considered. In that mode, work inside the model is split more in parallel across GPUs, which can improve throughput when the GPUs have a fast interconnect. In initial tests it was faster on this machine. However, tensor splitting requires more frequent communication between the cards, and the local setup could not guarantee high inter-GPU bandwidth because the second GPU slot had substantially fewer PCIe lanes than the primary slot. The `llama.cpp` documentation also marks tensor split as experimental and notes that it is more sensitive to GPU interconnect speed. [7]

For that reason, the final benchmark used layer splitting. This was a conservative deployment choice. It was slower in initial testing, but it reduced dependence on heavy inter-GPU communication and was judged more appropriate for long-running benchmark execution, where stability is key.

B Local ALFWorld Setup script and Task Subset

The local setup script prepared the ALFWorld evaluation subset before benchmark execution. It recreated the custom subset directory, updated the ALFWorld configuration files to use this subset,

set the visual resolution to 1280×720, and applied a small number of local compatibility fixes needed for the visual oracle setup.

Two ALFWorld compatibility issues were fixed locally. First, the selected `receps.json` files, which store information about receptacles in the scene, were missing the field that records whether a receptacle is initially closed. This was an issue because the visual oracle later expects this information when describing and interacting with receptacles. The missing values were therefore filled in during setup. Second, the oracle instance-segmentation code had two row/column variables swapped. This only became a problem after switching to a non-square 720p visual resolution, where the wrong loop order produced indexing errors. The variables were swapped back to the correct order.

Initial visual-mode testing also showed that some maps that were valid in text mode could not be completed in the local visual oracle setup. The clearest issue occurred in fridge-based tasks. After the agent picked up the target object and attempted to navigate to the fridge, visual mode returned the error `Cannot teleport due to hand object collision.`. The visual rendering placed the held object in front of the agent, so the carried object could collide with the fridge during the `go to` action. In principle the agent could:

1. **Drop** the object elsewhere.
2. **Navigate** to the fridge and open it.
3. **Return** and retrieve the object.
4. **Navigate back** to the open fridge.

This sequence was considered too complicated for the benchmark, and no agent would be expected to solve this within the maximum step limit, so these maps had to be excluded. After manually trying the maps, a task which uses a mug was found, whose collision geometry was small enough in testing that the task could be completed. Figure 10 shows the agent standing in front of the fridge while holding the mug in the final selected cooling map. The `look_at_obj_in_light` tasks also showed visual-oracle-specific failures, so a known-working map was manually selected for that task family as well.

The final subset was therefore selected deterministically rather than randomly. Candidate map folders were first filtered to ensure that the required files were present and that `game.tw-pddl` marked the task as solvable. The remaining selection was then constrained by manual visual-mode testing. Table 5 lists the exact trials that were used.

Task type	Selected trial	Scene
look_at_obj_in_light	look_at_obj_in_light-Bowl-None-DeskLamp-308/trial_T20190907_133919_856963	308
pick_and_place_simple	pick_and_place_simple-Mug-None-Desk-308/trial_T20190908_125200_737896	308
pick_clean_then_place_in_recep	pick_clean_then_place_in_recep-Bowl-None-Cabinet-10/trial_T20190909_061130_844814	10
pick_cool_then_place_in_recep	pick_cool_then_place_in_recep-Mug-None-Cabinet-10/trial_T20190909_121559_082363	10
pick_heat_then_place_in_recep	pick_heat_then_place_in_recep-Apple-None-Fridge-10/trial_T20190906_182259_116320	10
pick_two_obj_and_place	pick_two_obj_and_place-CD-None-Safe-308/trial_T20190907_050942_897916	308

Table 5: Exact ALFWorld trials used in the custom six-task subset.

Figure 10: Visual-mode view of the agent standing in front of the fridge while holding the mug in the selected cooling task.

C Local Stability Issues

Early local runs exposed two separate stability issues in the local execution stack. The first was a WSL-side memory issue during visual-mode startup, where ALFWorld, AI2-THOR, Unity, and the WSL graphical stack had to run alongside the Windows-side model server. The second was Windows-side memory issue during `llama.cpp` inference for a Gemma-family model.

C.1 WSL and visual-environment Memory Issues

One failed visual run completely exhausted memory and the pagefile, with the issue most likely stemming from Unity/AI2-THOR. This was mitigated by running the text and visual benchmarks separately and by increasing the WSL allocation to 20 GB of memory and 16 GB of swap.

C.2 Memory Issues With Gemma-4 Models

A separate issue occurred with `gemma-4-31b-it`, where llama.cpp crashed during inference. After inspecting the local logs, the failure appeared to be caused by memory exhaustion: RAM and page memory were both heavily depleted, with `llama-server.exe` reaching roughly 50 GB of memory use (not including the memory of the language model residing in VRAM). This was not normal behaviour and only caused an issue with this specific model. Online research showed that this was not an isolated issue, other users had reported similar Gemma-family memory problems in llama.cpp, regarding prompt cache and context checkpoints.
Sources:

https://github.com/ggml-org/llama.cpp/discussions/21480
https://github.com/ggml-org/llama.cpp/issues/21690

The local llama.cpp launcher was therefore changed to use:

```
--cache-ram 0
--ctx-checkpoints 1
--predict 8192
```

The first two settings disabled the in-RAM prompt cache and limited context checkpoints to one. These were added to reduce the Gemma-family memory pressure.

The `--predict 8192` setting served a different purpose: it limited server-side generation length after `gemma-4-12b-it` produced responses that did not stop naturally and continued until the 64K context window was exhausted. The cap prevented such failures from stalling the benchmark for an excessive amount of time. These settings were backend stability settings and should not be interpreted as changes to the ALFWorld tasks, prompts, tool schema, or model-facing observations.

D Local Generation Sampling Parameters

Table 6 reports the effective sampler settings for the local llama.cpp-served GGUF models. These values are included for reproducibility. The benchmark did not override sampler parameters in the OpenAI-compatible request or through llama.cpp launcher flags, so the effective values came from GGUF metadata where available and from llama.cpp defaults otherwise.

Model	GGUF file	Temp.	Top-p	Top-k	Source
qwen3.6-35b-a3b	Qwen3.6-35B-A3B-UD-Q4_K_XL.gguf	1.0	0.95	20	GGUF metadata
qwen3.6-27b	Qwen3.6-27B-UD-Q4_K_XL.gguf	1.0	0.95	20	GGUF metadata
qwen3.5-9b	Qwen3.5-9B-Q8_0.gguf	0.8	0.95	40	llama.cpp default
gemma-4-e4b-it	gemma-4-E4B-it-Q8_0.gguf	1.0	0.95	64	GGUF metadata
gemma-4-26b-a4b-it	gemma-4-26B-A4B-it-UD-Q4_K_XL.gguf	1.0	0.95	64	GGUF metadata
gemma-4-31b-it	gemma-4-31B-it-UD-Q4_K_XL.gguf	1.0	0.95	64	GGUF metadata
gemma-4-12b-it	gemma-4-12b-it-UD-Q4_K_XL.gguf	1.0	0.95	64	GGUF metadata

Table 6: Effective local sampler settings for the llama.cpp-served GGUF models. The benchmark request itself did not override these parameters.

These values describe the effective local runtime configuration, not necessarily the upstream provider-recommended sampling configuration.

E Result JSON Files

Each completed episode was saved as one JSON file. The file contains three main sections: *metadata*, *conversation* and *summary*. The metadata records the episode-level outcome, the conversation stores the turn-by-turn interaction and the summary stores aggregate timing, token, and power measurements.
A shortened version of a .json file from the results:

```
{
"metadata": {
"model_name": "qwen3.6-35b-a3b",
"mode": "text",
"success": true,
"step_count": 6,
"turn_count": 6,
"game_index": 1,
"episode_stop_reason": "success",
"tool_call_enabled": true,
"gpu_power_tracking": "dual"
},
"conversation": [
{
"turn": 4,
"step": 4,
"observation": "You arrive at fridge 1. The fridge 1 is closed.",
"tool_calls": [
{
"name": "cool",
"arguments": {
"object": "mug 3",
"with": "fridge 1"
}
}
],
"oracle_command": "cool mug 3 with fridge 1",
"tool_results": [
"You cool the mug 3 using the fridge 1."
],
"stats": {
"input_tokens": 2752,
"output_tokens": 88,
"wall_clock_time_ms": 1249.14,
"avg_gpu_power_watts": 149.56
},
"image_path": null
}
],
"summary": {
"total_input_tokens": 16274,
"total_output_tokens": 583,
"total_wall_clock_time_sec": 8.92,
"avg_tokens_per_sec": 65.37,
"avg_gpu_power_watts_overall": 147.62
}
}
```


F System Prompts

The prompts list the available actions for readability, but the executable tool interface was also provided to the models as an OpenAI-compatible function-tool schema with typed JSON arguments.

F.1 Text Mode Prompt

```
You are an AI agent interacting with a text-based household environment.
You will receive a text description of your current state.
You must output exactly ONE tool call per response.

## Available Tools

You must use tool calls to interact with the environment. Object IDs come from the observation text
(e.g., "mug 1", "desk 2").

1. go_to(receptacle="desk 2") - Move to a receptacle
2. look() - Look around your current location
3. inventory() - Check what you are holding
4. open(receptacle="fridge 1") - Open a receptacle
5. close(receptacle="fridge 1") - Close a receptacle
6. take(object="mug 1", from="desk 2") - Take an object from a receptacle
7. move(object="mug 1", to="shelf 1") - Move an object to a receptacle
8. examine(target="desk 2") - Examine a receptacle or object
9. use(object="desklamp 1") - Use an object (turn on lamp, etc.)
10. heat(object="mug 1", with="microwave 1") - Heat an object
11. clean(object="mug 1", with="sinkbasin 1") - Clean an object
12. cool(object="tomato 1", with="fridge 1") - Cool an object
13. slice(object="tomato 1", with="knife 1") - Slice an object

## CRITICAL OPERATIONAL RULES:
 1. State Tracking: Before each action, explicitly track your state in your thinking. Never assume
you are carrying an item unless you just successfully took it.
 2. Single Item Inventory: You can hold only 1 item at a time. Before you can ‘take‘ another object,
you must first ‘move <OBJECT> to <RECEPTACLE>‘ to drop what you are currently carrying.
 3. Action Preconditions & Syntax:
    - You MUST ‘take <OBJECT> from <RECEPTACLE>‘ before you can clean, move, use, or heat it.
    - Object names must match the observation EXACTLY (including numbers, e.g., ‘sinkbasin 1‘, not
‘sinkbasin‘).
    - ‘clean‘ requires a washable receptacle (sinkbasin, bathtubbasin). ‘move‘ requires a valid storage
receptacle (shelf, cart, etc.).
    - ‘go to‘ may fail with "Cannot teleport due to hand object collision" while holding an object.
If this happens, ‘move <OBJECT> to <RECEPTACLE>‘ first, then ‘go to‘ again.
    - ‘heat‘ requires the microwave to be **closed**. If open, ‘close <MICROWAVE>‘ first, then ‘heat‘.
    - ‘cool‘ requires the fridge to be **closed**. If open, ‘close <FRIDGE>‘ first, then ‘cool‘.
    - ‘slice‘ requires a **Knife** in your inventory. Pick up a knife first, then issue
    ‘slice <visible_object> with knife <N>‘.
    - ‘heat‘, ‘cool‘, and ‘clean‘ silently fail ("Nothing happens.") if their target receptacle is
not visible. Always verify the receptacle state before using these commands.
 4. Error Recovery: If an action returns "Nothing happens.", STOP repeating it immediately. Re-evaluate:
(a) Are you at the correct receptacle? (b) Do you have the required object in your inventory? (c)
Is the target receptacle valid for this action?
 5. Systematic Search: If the target object is not visible, systematically check likely receptacles
using ‘go_to()‘.

## Additional Information
You need to go to a receptacle before you can interact with it.
To do anything with an object, you first need to take it. To take it you first need to find it.
```

F.2 Visual Mode Prompt

```
You are an AI agent interacting with a 3D household environment.
You will receive an image of your current view along with a text description.
You must output exactly ONE tool call per response.

## Available Tools

You must use tool calls to interact with the environment. Object IDs come from the observation text
(e.g., "mug 1", "desk 2").

1. go_to(receptacle="desk 2") - Move to a receptacle
2. look() - Look around your current location
3. inventory() - Check what you are holding
4. open(receptacle="fridge 1") - Open a receptacle
5. close(receptacle="fridge 1") - Close a receptacle
6. take(object="mug 1", from="desk 2") - Take an object from a receptacle
7. move(object="mug 1", to="shelf 1") - Move an object to a receptacle
8. examine(target="desk 2") - Examine a receptacle or object
9. use(object="desklamp 1") - Use an object (turn on lamp, etc.)
10. heat(object="mug 1", with="microwave 1") - Heat an object
11. clean(object="mug 1", with="sinkbasin 1") - Clean an object
12. cool(object="tomato 1", with="fridge 1") - Cool an object
13. slice(object="tomato 1", with="knife 1") - Slice an object

## CRITICAL OPERATIONAL RULES:
 1. State Tracking: Before each action, explicitly track your state in your thinking. Never assume
you are carrying an item unless you just successfully took it.
 2. Single Item Inventory: You can hold only 1 item at a time. Before you can ‘take‘ another object,
you must first ‘move <OBJECT> to <RECEPTACLE>‘ to drop what you are currently carrying.
 3. Action Preconditions & Syntax:
    - You MUST ‘take <OBJECT> from <RECEPTACLE>‘ before you can clean, move, use, or heat it.
    - Object names must match the observation EXACTLY (including numbers, e.g., ‘sinkbasin 1‘, not
‘sinkbasin‘).
    - ‘clean‘ requires a washable receptacle (sinkbasin, bathtubbasin). ‘move‘ requires a valid storage
receptacle (shelf, cart, etc.).
    - ‘go to‘ may fail with "Cannot teleport due to hand object collision" while holding an object.
If this happens, ‘move <OBJECT> to <RECEPTACLE>‘ first, then ‘go to‘ again.
    - ‘heat‘ requires the microwave to be **closed**. If open, ‘close <MICROWAVE>‘ first, then ‘heat‘.
    - ‘cool‘ requires the fridge to be **closed**. If open, ‘close <FRIDGE>‘ first, then ‘cool‘.
    - ‘slice‘ requires a **Knife** in your inventory. Pick up a knife first, then issue
    ‘slice <visible_object> with knife <N>‘.
    - ‘heat‘, ‘cool‘, and ‘clean‘ silently fail ("Nothing happens.") if their target receptacle is
not visible. Always verify the receptacle state before using these commands.
 4. Error Recovery: If an action returns "Nothing happens.", STOP repeating it immediately. Re-evaluate:
(a) Are you at the correct receptacle? (b) Do you have the required object in your inventory? (c)
Is the target receptacle valid for this action?
 5. Systematic Search: If the target object is not visible, systematically check likely receptacles
using ‘go_to()‘.

## Additional Information
You need to go to a receptacle before you can interact with it. Even if something is in your field
of vision you still need to go to the receptacle first.
To do anything with an object, you first need to take it. To take it you first need to find it.
```