POLITECNICO DI TORINO
Repository ISTITUZIONALE

AgentQuest: A Modular Benchmark Framework to Measure Progress and Improve LLM Agents

Original
AgentQuest: A Modular Benchmark Framework to Measure Progress and Improve LLM Agents / Gioacchini, Luca; Siracusano, Giuseppe; Sanvito, Davide; Gashteovski, Kiril; Friede, David; Bifulco, Roberto; Lawrence, Carolin. - ELETTRONICO. - 3:(2024), pp. 185-193. (Intervento presentato al convegno 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies tenutosi a Mexico City (Mexico) nel June 16-21, 2024) [10.48550/arxiv.2404.06411].

Availability:
This version is available at: 11583/2989709 since: 2024-06-19T16:08:55Z

Publisher:
Association for Computational Linguistics

Published
DOI:10.48550/arxiv.2404.06411

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

02 January 2025

AgentQuest: A Modular Benchmark Framework to Measure Progress and Improve LLM Agents

Luca Gioacchini[1,2], Giuseppe Siracusano[1], Davide Sanvito[1], Kiril Gashteovski[1,3], David Friede[1], Roberto Bifulco[1], Carolin Lawrence[1]
[1] NEC Laboratories Europe, Heidelberg, Germany
[2] Politecnico di Torino, Turin, Italy
[3] CAIR, Ss. Cyril and Methodius University, Skopje, North Macedonia

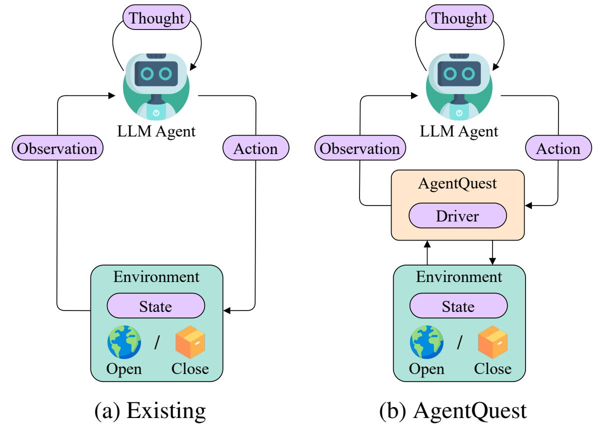
Abstract

The advances made by Large Language Models (LLMs) have led to the pursuit of LLM agents that can solve intricate, multi-step reasoning tasks. As with any research pursuit, benchmarking and evaluation are key corner stones to efficient and reliable progress. However, existing benchmarks are often narrow and simply compute overall task success. To face these issues, we propose AgentQuest[1] – a framework where (i) both benchmarks and metrics are modular and easily extensible through well documented and easy-to-use APIs; (ii) we offer two new evaluation metrics that can reliably track LLM agent progress while solving a task. We exemplify the utility of the metrics on two use cases wherein we identify common failure points and refine the agent architecture to obtain a significant performance increase. Together with the research community, we hope to extend AgentQuest further and therefore we make it available under https://github.com/nec-research/agentquest.

1 Introduction

Generative Agents (Kiela et al., 2023) are software systems that leverage foundation models like Large Language Models (LLMs) to perform complex tasks, take decisions, devise multi-steps plans and use tools (API calls, coding, etc.) to build solutions in heterogeneous contexts (Wang et al., 2023; Weng, 2023). The potential ability to solve heterogeneous tasks with high degrees of autonomy has catalysed the interest of both research and industrial communities. Nonetheless, it is still unclear to which extent current systems are successfully able to fulfil their promises. In fact, methodologies to benchmark, evaluate and advance these systems are still in their early days.

We identify a couple of gaps. Firstly, benchmarking agents requires combining different benchmark types (Liu et al., 2023; Chalamalasetti et al., 2023). For example, some benchmarks focus on specific capabilities and provide gaming environments, which we refer to as "closed-box" – i.e. with a finite set of actions (Liu et al., 2023; Patil et al., 2023; Chalamalasetti et al., 2023) – whereas other benchmarks provide open-ended tasks and access to general tools, like web browsing (Zhuang et al., 2023; Zheng et al., 2023; Mialon et al., 2023). As benchmarks are developed independently, significant effort goes into custom integration of new agent architectures with each benchmark.



(a) Existing (b) AgentQuest

Figure 1: Overview of agent-benchmark interactions in existing frameworks and in AgentQuest. AgentQuest defines a common interface to interact with the benchmarks and to compute progress metrics, easing the addition of new benchmarks and allowing researchers to evaluate and debug their agent architectures.

Secondly, and more critically, existing benchmarks mostly focus on providing a *success rate* measure, i.e. a binary success/fail evaluation for each of the proposed tasks. While success rate is helpful to measure overall advances of an agent technology, it has limited use in guiding improvements for new generative agent architectures. Here, it is important to consider that generative agents often combine foundation models with multiple other components, such as memory and tools. Develop-

[1] Demo provided at https://youtu.be/0JNkIfwnoak.

Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 3: System Demonstrations), pages 185–193
June 16-21, 2024 ©2024 Association for Computational Linguistics

ers can reason about these individual components in terms of architecture and their inter-dependence, and could actively change and evolve them using deeper insights about how an agent performs in a benchmark. That is, developers need benchmarks to both evaluate and *debug* agents.

For example, current benchmarks make it hard to answer questions like *does the agent fail completely the tasks or does it partially solve them? Does the agent fail consistently at a certain step? Would extra run time lead to finding a solution?* Answering these questions would require tracing and inspecting the execution of the agent. We argue that providing a more efficient approach that is consistent over multiple benchmarks is a stepping stone towards evolving generative agents.

We address these gaps introducing AgentQuest, a modular framework to support multiple diverse benchmarks and agent architectures (See Figure 1), alongside with two new metrics – i.e. progress rate and repetition rate – to debug an agent architecture behaviour. AgentQuest defines a standard interface to connect an arbitrary agent architecture with diverse benchmarks, and to compute progress and repetition rates from them.

We showcase the framework, implementing 4 benchmarks in AgentQuest: ALFWorld (Shridhar et al., 2020), Lateral Thinking Puzzles (Sloane, 1992), Mastermind and Sudoku. The latter two are newly introduced with AgentQuest. Additional benchmarks can be easily added, while requiring no changes to the tested agents.

Our final contribution is to present our experience leveraging the proposed metrics to debug and improve existing agent architectures as implemented in LangChain (Chase, 2022). In particular, we show that in the Mastermind benchmark the combination of progress rate and repetition rate identifies a limitation in the ability of the agent to explore the full space of potential solutions. Guided by this insight we could improve the success rate in this benchmark by up to ≈20%. In Lateral Thinking Puzzles we show that partially repeating actions is part of the agent strategy, whereas in ALFWorld, we show that monitoring the progress rate makes it possible to identify that the final success rate is limited by the allowed runtime of the agent, and that more steps lead to a better performance. Finally, in the Sudoku benchmark, we show that the low success rate is actually paired with low progress rate, making clear that the tested agent is unable to solve this type of tasks.

2 Generative AI Agents in a Nutshell

Generative AI agents are automated systems relying on software components integrated with LLMs pre-trained on large amount of data for language understanding and processing. When assigned a task, an agent engages in a systematic process: it iteratively formulates self-generated instructions, executes them, and observes the outcomes until the ultimate objective is achieved. Next, we showcase the basic interaction between agents and the environment in which they operate and describe the standard benchmarking techniques.

2.1 Agent-Environment interaction

Closely following the terminology in Reinforcement Learning (RL)[2] (Sutton and Barto, 2018), the core elements defining the agent-environment interaction are *environment*, *state*, *observation* and *action* (see Figure 1a).

Environment and states. The environment refers to the external system the agent interacts with. In this context, we treat the benchmark and the environment as synonyms. It is typically described through a finite set of hidden *states*, which are not directly observable by the agent and represent the benchmark configuration.

Observations and actions. The agent interacts with the environment for multiple execution steps. At each step, the environment produces an *observation* providing information about its current hidden state. The agent uses the internal LLM to process the received observation. Being pre-trained on general knowledge data, the LLM engages a reasoning process generating a *thought* on the observation (e.g. the planned strategy to follow in the current step or the usage of a tool). According to this thought, the agent provides the environment an *action* to modify the current hidden state. [3]

The following is an example of an agent interacting with a closed-box environment to solve a code-breaking challenge, i.e. the Mastermind game (Stuckman and Zhang, 2005). The agent is tasked to guess a four-digit code. At each step, the agent devises a strategy and acts accordingly to guess the code. The environment provides a feedback on the number of guessed digits in correct

[2] Unlike RL scenarios, the agent does not need a further training process. It relies on the pre-trained LLM and does not perform an action under the influence of any reward.

[3] Unlike RL, the LLM outputs are unconstrained, and any provided action is considered valid.

and wrong positions. The process continues until a stopping condition is met.

```
ENVIRONMENT: Initializing a new environment...
OBSERVATION: Start guessing the 4 digits code.
THOUGHT: I'll start by guessing 1234 to get an idea of which
of these numbers are in the secret number and if any are in
the correct position.
ACTION: 1234

ENVIRONMENT: Comparing the guessed code with the truth...
OBSERVATION: Your guess has 1 correct numbers in the wrong
position and 0 correct numbers in the correct position.
Keep guessing...
THOUGHT: One of the digits 1, 2, 3, or 4 is correct but in
the wrong position. I'll try to move each digit to a
different position to see if I can get a correct position.
ACTION: 2143
...
```

2.2 Benchmarking an agent

When evaluating agents performance on a benchmark, the following two metrics are commonly used (Liu et al., 2023): (i) Success Rate (SR), i.e. the ratio of successful tasks to the total instances; (ii) Time to Success, i.e. the average time required to obtain a solution. While important and trending metrics (Chalamalasetti et al., 2023; Hessel et al., 2022; Zhang et al., 2020a), they exclusively address the final success. They cannot measure intermediate success or failure and therefore make it difficult to understand why agents might systematically fail and how they can be improved. In contrast, we want to define intermediate metrics that allow us to easily assess and compare the performance of agents across a wide range of tasks.

3 AgentQuest Overview

We designed AgentQuest as a separation layer between agent and environment (see Figure 1b). Essentially, it offers (i) a unified interface (i.e. the *driver*) ensuring compatibility between different agent architectures and benchmarks with minimal programming efforts (Section 3.1); (ii) the implementation of two metrics beyond task success (i.e. *progress rate* and *repetition rate*) aimed at monitoring the agent advancement toward the final goal and allowing us to understand the reasons behind failures (Section 3.2); (iii) a unique vantage point and interface for implementing new metrics to monitoring and measuring the execution (Section 3.3).

3.1 Benchmarks common interface

Different benchmarks require invoking distinct functions, using specific formats, and performing parsing and post-processing of observations and agent actions. To integrate different agent architectures, the common trend is hardcoding such benchmark-specific requirements directly in the framework (Liu et al. 2023; Chalamalasetti et al. 2023, *inter alia*). This results in many custom interfaces tailored on each environment, making it difficult to easily move to other benchmarks and agent architectures.

Instead, AgentQuest exposes a single unified Python interface, i.e. the `Driver` and two classes reflecting the agent-environment interaction components (i.e. `Observation`, `Action`).

Observations and actions. We provide two simple classes: `Observation` and `Action`. The first has two required attributes: (i) `output`, a string reporting information about the environment state; (ii) done, a Boolean variable indicating if the final task is currently accomplished or not. The `Action` class has one required attribute, `action_value`. It is a string directly output by the agent. Once processed and provided to the environment, it triggers the environment change. To customise the interactions, developers can define optional attributes.

Driver. We provide the `Driver` class with two mandatory methods: (i) the `reset` method initialises a new instance of the environment and returns the first observation; (ii) the `step` method performs one single execution step. It accepts one instance of the `Action` class from the agent, processes the action (e.g. parses the `action_value` string) and uses it to modify the environment state. It always returns an observation. The driver supports also the benchmark-specific `state` attribute, acting as a simple API. It exposes the environment state at step t, useful to compute the progress rate.

We here provide an example of the implemented interaction for Mastermind:

```python
from agentquest.drivers import MasterMindDriver
from agentquest.utils import Action
from agentquest.metrics import get_progress, get_repetition

agent = ... # Initialize your agent
actions, progress, repetitions = [], [], []
# Initialize the environment and reset round
driver = MasterMindDriver(truth='5618')
obs = driver.reset()
# Agent loop
while not obs.done:
    guess = agent(obs.output) # Get the agent output
    action = Action(action_value=guess) # Create action
    actions.append(action.action_value) # Store action
    obs = driver.step(action) # Execute step
    # Compute current progress and repetition
    progress.append(get_progress(driver.state, '5618'))
    repetitions.append(get_repetitions(actions))
    # Extend with your custom metrics here ...
# Compute final metrics
PR = [x/len('5618') for x in progress]
RR = [x/(len(actions)-1) for x in repetitions]
```

3.2 Understanding agent advancements

Getting insights on how they tackle a specific task is key to comprehend agent behaviours, capabilities and limitations. Furthermore, identifying systematic agent failures allows to pinpoint necessary adjustments within the architecture to effectively address the underlying issues.

AgentQuest contributes towards this direction introducing two cross-benchmark metrics, the *progress rate* and the *repetition rate*. While the first expresses *how much* the agent is advancing towards the final goal, the latter indicates *how* it is reaching it, with a specific focus on the amount of repeated (i.e. similar) actions the agent performs.

Milestones and progress rate. To quantify the agent advancement towards the final goal, AgentQuest uses a set of *milestones* $\mathcal{M}$. In a nutshell, we break down the final solution into a series of environment hidden states the agent needs to reach to get the final solution of the task, hence, $\mathcal{M} \subseteq \mathcal{S}$, where $\mathcal{S}$ is the set of hidden states. The magnitude of $\mathcal{M}$ determines the level of *granularity* in the evaluation process. Specifically, when $\mathcal{M}$ aligns closely with $\mathcal{S}$, it offers a more comprehensive insight into the agent progress, resulting in finer granularity, whereas for $|\mathcal{M}| = 1$ the evaluation coincides with the success rate.

We assign a score to all the states included in $\mathcal{M}$ through a scoring function f and, at execution step t, we define the *progress rate* $\text{PR}_t : \mathcal{S} \rightarrow [0, 1]$ dependant of such scoring function, as an indication of how far the agent is from the goal, allowing to track agent progress over time. Depending on the benchmark, the progress rate might also decrease during the execution. Milestones can either be manually annotated, or internally computed.

Repetition rate. The repetition rate RR_t is a measure of the agent tendency of repeating actions. Depending on the benchmark, we do not consider repetitions as a limitation, – e.g. solving a maze requires repetitions, such as going left repeatedly. See also Section 4 for a positive and negative example of repetitions.

At execution step t, we consider the set of unique actions taken by the agent up to $t-1$, $\mathcal{A}_{t-1}$. Then, we compute the similarity function g between the current action a_t and all the previous ones in $\mathcal{A}_{t-1}$. As any action generated by the LLM is considered valid, we consider the action a_t as *repeated* if it exists at least one previous action $a \in \mathcal{A}_{t-1}$ such that $g(a_t, a) \geq \theta_a$, where $\theta_a \in [0, 1]$ is the *resolution*.[4] If the action is not repeated, we update the set of unique actions as $A_t = A_{t-1} \cup a_t$.

Based on this, we define the repetition rate at step t as the cumulative number of repeated actions normalised by the number of execution steps, T, except for the first. Formally, $\text{RR}_t = \frac{t-|A_t|}{T-1}$.

Table 1: Attributes exposing components of the agent-environment interaction useful to define new metrics.

Class	Attribute	Access to
`Driver`	`state`	Hidden states
`Observation`	`output`	Observations
`Action`	`action_value`	Agent actions

3.3 Adding new metrics

We rely on the progress and repetition rates to show how AgentQuest can be extended with new metrics through a simple function template. We then show the implementations of the functions adapted to the considered benchmark.

Metric function template. We use a Python function template to easily define the elements of the agent-environment interactions required for computing a given metric. Table 1 provides a recap of the main attributes and reference classes that can be used as input for the custom metrics. Additionally, users can provide external data, like milestones or action history.

Implement progress rate. Depending on the benchmark, developers need to implement the custom scoring function f through the `get_progress` function and define the set of milestones $\mathcal{M}$. Milestones can either be user-defined or internally computed within `get_progress`. Here, we show the definition of `get_progress` to quantify the achieved milestones for Mastermind. The milestones are the digits of the final solution and the progress indicates the count of correctly guessed digits in their positions:

```
def get_progress(state, milestones):
    reached_milestones = 0 # Digits in correct position
    for i, j in zip(state, milestones):
        if i == j: reached_milestones += 1
    return reached_milestones

# Usage example. The code to guess is '5618'
progress = get_progress('2318', '5618') # Reached milestones
>>> 2
progress/len('5618') # Compute Progress Rate
>>> 0.5
```

[4] A higher resolution demands closer matches for classification as repeated actions, while lower values broaden the spectrum of qualifying action similarities.

Table 2: Overview of the benchmarks provided in AgentQuest.

Benchmark	Description	Milestones
Mastermind	Guessing a numeric code with feedback on guessed digits and positions.	Digits of the code to guess.
LTP	Solving riddles by asking Yes/No questions.	Guessed riddle key aspects.
ALFWorld	Finding an object in a textual world and using it.	Sequence of actions.
Sudoku	9x9 grid puzzle. Digits 1-9 fill each column, row, and 3x3 sub-grid without repetition.	Total number of correct inserted digits.

Implement repetition rate. To determine if an action is repeated, the end user must define the similarity function g according to the considered benchmark. We provide the `get_repetitions` template function to compute the number of repeated actions. Here, we illustrate its implementation in Python and provide a usage example for Mastermind, where g is the Levenshtein similarity (Levenshtein, 1966).

```
from Levenshtein import ratio as g

def get_repetitions(actions, THETA_A):
    unique_act = set() # Initialise unique actions
    for i,a in enumerate(actions):
        # Check for repetitions
        if all([g(a,actions[x])<THETA_A for x in range(i)]):
            unique_act.add(a)
    return len(actions)-len(unique_act)

# Usage example. The code to guess is '5618'
actions = ['1234', '2143', '1234', '5618'] # Actions history
repetitions = get_repetitions(actions, 1.0)
>>> 1 repeated action
# Compute Repetition Rate
repetitions/(len(actions)-1)
>>> 0.33
```

In other cases, where a can be any text string, we can use standard metrics, such as BLEU (Papineni et al., 2002), ROGUE (Lin, 2004) or BERTScore (Zhang et al., 2020b).

4 Insights via AgentQuest

We investigate agent behaviours in different reasoning scenarios by proposing a starting set of four benchmarks. We implemented from scratch Sudoku (Felgenhauer and Jarvis, 2006) and Mastermind (Stuckman and Zhang, 2005) environments, while ALFWorld (Shridhar et al., 2020) and Lateral Thinking Puzzles (LTP)(Sloane, 1992) are existing implementations (Liu et al., 2023). Table 2 provides an overview of the benchmarks and their respective milestones used to measure progress.

We emphasise that this evaluation is not aimed at providing a thorough evaluation and comparison of agent architectures, but rather to show how to use AgentQuest and how monitoring progress and action repetition can provide relevant insights to developers, even after a few executions.

Table 3: Average existing and proposed metrics for the tested benchmarks. We report the metrics, Success Rate (SR), Steps, Progress Rate at step 60 (PR_{60}) and Repetition Rate at final step 60 (RR_{60}). We denote with * the improved results after modifying the agent architecture.

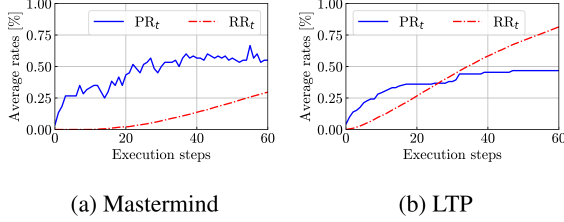
	Existing Metrics		AgentQuest	
	SR	**Steps**	**PR_{60}**	**RR_{60}**
Mastermind	0.47	41.87	0.62	0.32
LTP	0.20	52.00	0.46	0.81
ALFWorld	0.86	21.00	0.74	0.06
Sudoku	0.00	59.67	0.08	0.22
Mastermind*	0.60	39.73	0.73	0.00
ALFWorld*	0.93	25.86	$0.80^\dagger$	$0.07^\dagger$

†Metrics referred to the extended runtime up to 120 steps, hence PR_{120} and RR_{120}.

Experimental setup. We use as reference architecture the off-the-shelf chat agent provided by LangChain (Chase, 2022) powered by GPT-4 (OpenAI, 2023b) as LLM because it is intuitive, easy to extend and open source. We run 15 instances of the four benchmarks within AgentQuest, setting the maximum number of execution steps as 60[5]. In Appendix B we provide examples on how to use AgentQuest with two additional agent architectures and GAIA (Mialon et al., 2023) as open-ended environment.

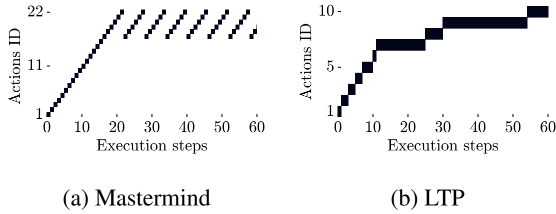
Experimental results. For Mastermind, Figure 2a shows the progress rate PR_t and repetition rate RR_t. In the first 22 steps, the agent explores different solutions ($RR_{[0,22]} < 5\%$). This leads to growing progress towards the final goal, reaching half of the milestones ($PR_{22} \approx 55\%$). Then, the agent starts performing the same actions, exhibiting a repetitive pattern (see also Figure 3a rightmost part) and failing to reach the final goal within the

[5]We limit the number of instances in our experiments for two main reasons: (i) the work primarily serves as a demonstration of the developed framework itself, rather than an extensive evaluation of the agent performance; (ii) extensive tests could have significantly impacted the ability to reproduce the experiments due to the expensive nature of API calls.



(a) Mastermind (b) LTP

Figure 2: Average Progress rate PR$_t$ and the repetition rate RR$_t$ on Mastermind and LTP. Mastermind: It starts out with a low RR$_t$ but this increases after step 22 while the progress rate also stall at 55%. LTP: at first a higher RR$_t$ allows the agent to progress by making small variations that lead to success, but later this plateaus.



(a) Mastermind (b) LTP

Figure 3: Examples of repeated actions in Mastermind and LTP. Mastermind: there is a set of unique actions at first, but then gets stuck repeating the same actions over and over. LTP: repeated actions are small variations of the same question that lead to progress.

next 38 steps. This results in a rise of the repetitions to $\text{RR}_{60} = 30\%$ and a saturation of the progress rate at $\text{PR}_{60} = 55\%$. Hence, AgentQuest offered us a crucial insights on why the current agent cannot solve the Mastermind game.

To overcome this agent limitation we incorporate a memory component (Park et al., 2023) into the agent architecture. The agent stores the past guesses in a local buffer. Then, at each step, if the agent outputs an action already in the buffer, it is prompted to provide a new one. Table 3 (Mastermind*) shows that this simple change in agent architecture has a big impact: the agent can now solve more instances, increasing the final SR from 47% to 60% and preventing repetitions ($\text{RR}_{60} = 0\%$). This highlights how studying the interplay between progress and repetition rates can allow us to improve agent architecture, sometimes even with simple remedies. We support our intuition extending the evaluation to more instances of Mastermind from 15 to 60 achieving comparable results – i.e. 43% of SR with the standard architecture and 62% with the simple memory (19% of improvement).

For LTP, the AgentQuest metrics reveal a different agent behaviour, where repetitions are part of the agent reasoning strategy, enhancing the progress rate (Figure 2b). From the initial steps, the agent changes aspects of the same questions until a local solution emerges. This leads to horizontal indicators in Figure 3b and $\text{RR}_{20} \approx 30\%$. Despite solving only a few riddles (SR=0.2), these repetitions contribute to progress, achieving 46% of the milestones by the end of the execution, with a final repetition rate of $\text{RR}_{60} = 81\%$. This shows us how the interplay of progress and repetition rates provides an insight on how agents behave across the different time steps.

Consider the benchmark ALFWorld in Table 3 (we report the metrics trend in Appendix A). It requires the exploration of a textual world to locate an object. While the agent explores the solution space and limits action repetitions ($\text{RR}_{60} = 6\%$), it fails to solve all the games ($\text{PR}_{60} = 74\%$). This discrepancy may arise from the more exploration steps required to discover the object. We support this intuition extending the benchmark runtime to 120 steps resulting in a success and progress rates increase by 6% (ALFWorld* in Table 3). This confirms the usefulness of AgentQuest in understanding the agent failures.We support our intuition also extending the evaluation to more instances of ALFWorld from 15 to 60 achieving comparable results – i.e. 83% of SR with 60 steps as limit and 87% with 120 steps as limit (4% of improvement).

Finally, we look at Sudoku, known for its high level of difficulty (Felgenhauer and Jarvis, 2006). The low progress and repetition rates achieved after 60 steps ($\text{PR}_{60} = 8\%$ and $\text{RR}_{60} = 22\%$) indicate that the current agent architecture struggles in finding correct solutions solving this task. We report the metrics trend in Appendix A.

5 Conclusions

AgentQuest allows the research community to keep track of agent progress in a holistic manner. Starting out with a first set of four benchmarks and two new metrics, AgentQuest is easily extendable. Furthermore, the two proposed metrics, progress and repetition rates, have the great advantage of allowing to track how agents advance toward the final goal over time. Especially studying their interplay can lead to important insights that will allow the research community to improve agent performance. Finally, we believe that promptly sharing AgentQuest with the research community will fa-

cilitate benchmarking and debugging agents, and will foster the creation and use of new benchmarks and metrics.

Ethical Considerations

The complexity of LLM agents poses challenges in comprehending their decision-making processes. Ethical guidelines must demand transparency in such systems, ensuring that developers and end-users comprehend how decisions are reached.

We are not aware of any direct ethical impact generated by our work. However, we hope that insights into Generative AI agents' decision-making processes will be applied to improve and promote transparency and fairness.

Acknowledgements

This project has received funding from the European Union's Horizon Europe research and innovation programme (SNS-JU) under the Grant Agreement No 101139285 ("NATWORK").

References

Kranti Chalamalasetti, Jana Götze, Sherzod Hakimov, Brielen Madureira, Philipp Sadler, and David Schlangen. 2023. Clembench: Using Game Play to Evaluate Chat-Optimized Language Models as Conversational Agents.

Harrison Chase. 2022. LangChain - Building applications with LLMs through composability.

Bertram Felgenhauer and Frazer Jarvis. 2006. Mathematics of Sudoku I. *Mathematical Spectrum*.

Jack Hessel, Ari Holtzman, Maxwell Forbes, Ronan Le Bras, and Yejin Choi. 2022. CLIPScore: A Reference-free Evaluation Metric for Image Captioning.

Douwe Kiela, Tristan Thrush, Kawin Ethayarajh, and Amanpreet Singh. 2023. Plotting Progress in AI. *Contextual AI Blog*.

Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*.

Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2023. AgentBench: Evaluating LLMs as Agents.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. GAIA: a benchmark for General AI Assistants.

OpenAI. 2023a. Assistants API.

OpenAI. 2023b. GPT-4 Technical Report.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large Language Model Connected with Massive APIs.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2020. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning.

Paul Sloane. 1992. *Lateral Thinking Puzzlers*. Sterling Publishing Company, Inc.

Jeff Stuckman and Guo-Qiang Zhang. 2005. Mastermind is NP-complete.

Richard S Sutton and Andrew G Barto. 2018. *Reinforcement Learning: An Introduction*. MIT press.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2023. A Survey on Large Language Model based Autonomous Agents.

Lilian Weng. 2023. LLM-powered Autonomous Agents.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing Reasoning and Acting in Language Models.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020a. BERTScore: Evaluating Text Generation with BERT.

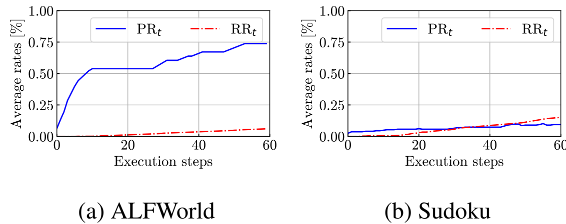
Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020b. BERTScore: Evaluating Text Generation with BERT.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. ToolQA: A Dataset for LLM Question Answering with External Tools.

A Appendix: ALFWorld and Sudoku benchmarks

In this section we report the detailed metrics for each step for the ALFWorld and Sudoku benchmarks, omitted for the sake of brevity from the main paper.



(a) ALFWorld (b) Sudoku

Figure 4: Progress rate PR_t and the repetition rate RR_t on ALFWorld and Sudoku averaged over 15 runs. ALFWorld: It starts out with a low repetition rate and quick increase of the progress rate. Then a slow increase of the repetition rate enables to further increase the progress rate although less quickly. Sudoku: The progress rate quickly reaches 8%. The repetition rate then slowly increases without any positive change in the progress rate.

Figure 4a reports the progress rate and repetition rate for ALFWorld. The repetition rate is close to 0% for the first 20 steps, then it slowly increases up to 6% after 60 steps. The progress rate quickly reaches over 50% in 10 steps, then keeps increasing, although slowly, up to 74%. The consistent improvement of the progress rate even for steps close to 60 together with the low repetition rate suggests that higher values may be reached by increasing the maximum number of steps. We validate this hypothesis by extending the benchmark runtime to 120 steps. As previously reported in Table 3, this results in an improvement of 6 percentage points for both the success rate the progress rate, i.e. SR$=93\%$ and $\mathrm{PR}_{120}=80\%$.

Figure 4b includes the two metrics for the Sudoku benchmark. We can observe that the progress rate quickly reaches a plateau at 8% in very few steps. The repetition rate is close to 0% for the first 10 steps, then it slowly increases up to 22% after 60 steps without any improvement of the progress rate.

B Appendix: Additional agents architectures and benchmarks

In this section we highlight the plug-and-play aspect of AgentQuest showing the implementation of Mastermind with two additional agents architectures, i.e. ReAct (Yao et al., 2022) as the most used architecture in literature and OpenAI Assistant (OpenAI, 2023a), as the most recent proprietary architecture. Additionally, we show how to implement the open-ended benchmark GAIA (Mialon et al., 2023) requiring the usage of external tools. For brevity, in the following snippets we omit details, like error handling or full agent definition. The complete code is available in the GitHub repository.

B.1 ReAct for Closed-box Environments

We show an example of how to execute a closed-box benchmark (i.e. ALFWorld) with an agent based on the ReAct architecture (Yao et al., 2022). Such architecture forces the agent decision making process to generate both textual reasoning traces and actions pertaining to a task in an interleaved manner. Common implementations (Chase, 2022; Yao et al., 2022) rely on external tools to perform actions. Here, we ensure compatibility with existing implementations providing a single tool (i.e. `ProxyTool`) that forwards the actions to the driver. In a nutshell, the agent reflects on the action to take and invokes the tool. Then, we feed the tool input to the driver to perform the interaction with the environment. At each step, we provide the agent the updated history of the actions and observations through the `intermediate_steps` variable.

```
from agentquest.drivers import MasterMindDriver
from agentquest.metrics import ...
from agentquest.utils import Action
...

# Define a dummy tool for closed-box environments
class ProxyTool(BaseTool):
    name = "proxytool"
    description = "Provide the action you want to perform"
    def _run(self):
        pass

# Instantiate custom prompt
prompt = CustomPromptTemplate(
    template=..., # LLM prompt
    tools=[ProxyTool()],
    input_variables=["intermediate_steps", ...]
)
# Initialise the agent
agent = create_react_agent(llm, [ProxyTool()], prompt)
intermediate_steps = []
# Initialise the driver
driver = MasterMindDriver(game)
# Get the first observation
obs = driver.reset()
# Agent Loop
while not obs.done:
    # Retrieve the agent output
    agent_choice = agent.invoke(
        {'input':obs.output,
         'intermediate_steps':intermediate_steps}
    )
    action = Action(action_value=agent_choice.tool_input)
    # Perform the step
    obs = driver.step(action)
    # Update intermediate steps
    intermediate_steps.append((agent_choice, obs.output))
    # Get current metrics ...
```

B.2 OpenAI Assistant for Closed-box Environments

The OpenAI Assistant (OpenAI, 2023a) is a proprietary architecture. It allows users to define custom agents by specifying the tasks to accomplish and the set of tools the agent can use. While the decision-making process is not directly accessible by the end-users (the agent and the LLM are hosted on the proprietary cloud environment), the tools can be invoked both remotely or locally. In the latter, users have control on the tool invocation managing the agent loop.

Similarly to ReAct, we here rely on the `ProxyTool`, acting as a proxy between the agent and the environment. We invoke the remote agent with the initial task (e.g. first ALFWorld observation) and process the output of its decision making process, i.e. the action to perform provided as tool input. Then, we bypass the tool invocation, directly forwarding the action to the driver to perform the execution step and retrieve the next observation. Finally, we invoke the agent with the new observation concluding the execution step.

```
from agentquest.drivers import MasterMindDriver
from agentquest.metrics import ...
from agentquest.utils import Action
...

# Define a dummy tool for closed-box environments
class ProxyTool(BaseTool):
    name = "proxytool"
    description = "Provide the action you want to perform"
    def _run(self):
        pass

# Initialise the agent
agent = OpenAIAssistantRunnable.create_assistant(
    instructions=... # LLM prompt
    tools=[ProxyTool()],
    model=... # Chosen LLM
    as_agent=True
)
# Initialise the driver
driver = MasterMindDriver(game)
# Get the first observation
obs = driver.reset()
# Get the first action
response = agent.invoke({"content": obs.output})
# Agent Loop
while not obs.done:
    # Retrieve the agent output
    agent_guess = response[0].tool_input
    action = Action(action_value=agent_guess)
    # Perform the step
    obs = driver.step(action)
    # Get current metrics ...
    # Manage Proxy Tool output
    tool_outputs = [
        {"output": obs.output,
         "tool_call_id": response[0].tool_call_id}
    ]
    # Invoke the agent to get the next action
    response = agent.invoke(
        {"tool_outputs": tool_outputs,
         "run_id": response[0].run_id,
         "thread_id": response[0].thread_id}
    )
```

B.3 OpenAI Assistant for Open-ended Environments

When interacting with an open-ended environment, the agent is not restricted to the pre-defined actions of the closed-box environment and it is allowed to select any user-defined tool (e.g. retrieving information online or executing code). Hence, we provide the agent the list of tools via the `tool` variable. The agent relies on its reasoning process to choose which tool to invoke. Omitted here for the sake of brevity, we rely of the manual annotations of the GAIA questions (Mialon et al., 2023) as milestones to compute the progress rate.

```
from agentquest.drivers import GaiaDriver
from agentquest.metrics import ...
from agentquest.utils import Action
...

# Define the tools
tools=[
    OnlineSearch(), # Retrieve a web page link
    WebContentParser(), # Read the web page
    FinalAnswerRetriever(), # Provide the final answer
    ...
]
# Initialise the agent
agent = OpenAIAssistantRunnable.create_assistant(
    instructions=... # LLM prompt
    tools=tools,
    model=... # Chosen LLM
    as_agent=True
)
# Initialise the driver
driver = GaiaDriver(question, tools)
# Get the first observation
obs = driver.reset()
# Get the first action
response = agent.invoke({"content": obs.output})
# Agent Loop
while not obs.done:
    # Retrieve the agent output
    act = f'{response[0].tool}:{response[0].tool_input}'
    action = Action(action_value=act)
    # Perform the step invoking the local tool
    obs = driver.step(action)
    # Get current metrics ...
    # Manage tool output as observation
    tool_outputs = [
        {"output": obs.output,
         "tool_call_id": response[0].tool_call_id}
    ]
    # Invoke the agent to get the next action
    response = agent.invoke(
        {"tool_outputs": tool_outputs,
         "run_id": response[0].run_id,
         "thread_id": response[0].thread_id}
    )
```

Here, the driver acts as a wrapper, executing the tool with the parameters provided by the agent (`tool_input`) and forwards the output to the agent in the correct format:

```
class GaiaDriver():
    def __init__(self, question, tools, ...):
        # Initialise the tool lookup
        self.tool_lookup = {x.name:x for x in tools}
    ...
    def step(self, action):
        # Parse the action
        tool, tool_input = action.action_value.split(':')
        # Invoke the tool
        tool_out = self.tool_lookup[tool]._run(tool_input)
        # Parse the tool output here ...
        return Observation(output=tool_out)
```