Titre: Title:	Standardisation et reproductibilité dans l'évaluation des agents web basés sur les grands modèles de langage
Auteur: Author:	Thibault Le Sellier de Chezelles
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Le Sellier de Chezelles, T. (2025). Standardisation et reproductibilité dans l'évaluation des agents web basés sur les grands modèles de langage [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/65950/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/65950/
Directeurs de recherche: Advisors:	Quentin Cappart, & Maxime Gasse
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Standardisation et Reproductibilité dans l'Évaluation des Agents Web Basés sur les Grands Modèles de Langage

THIBAULT LE SELLIER DE CHEZELLES
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Mai 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Standardisation et Reproductibilité dans l'Évaluation des Agents Web Basés sur les Grands Modèles de Langage

présenté par **Thibault LE SELLIER DE CHEZELLES**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Christopher J. PAL, président
Quentin CAPPART, membre et directeur de recherche
Maxime GASSE, membre et codirecteur de recherche
Nikolay RADOEV, membre

DÉDICACE

À tous mes co-stagiaires et amis,
qui ont dû supporter mes râleries lorsque les tests ne passaient pas,
vous me manquerez...

REMERCIEMENTS

Je tiens à exprimer ma profonde gratitude à toutes les personnes qui ont contribué à la réalisation de ce mémoire. Tout d'abord, Quentin et Maxime, mes superviseurs, pour leur encadrement exceptionnel, leurs précieux conseils et leur soutien constant tout au long de ce projet.

Ensuite, toute l'équipe UI Assist de ServiceNow Research, Allac, Massimo, Léo, Megh, Sahar, Emiliano, pour leur accueil et leur gentillesse.

Finalement, l'organisme Mitacs, dont les financements ont rendu possible cette maîtrise.

RÉSUMÉ

Les agents web basés sur les grands modèles de langage (*Large Language Models*, LLM) émergent comme une technologie prometteuse permettant d'automatiser des interactions complexes avec les interfaces web. Malgré leur potentiel élevé, l'évaluation rigoureuse de ces agents reste difficile en raison d'une fragmentation significative des benchmarks existants, de l'instabilité des environnements web, et de l'absence de standards méthodologiques clairement établis.

Ce mémoire introduit AgentLab, un cadre expérimental modulaire et extensible développé afin de répondre à ces défis méthodologiques. Intégré à l'écosystème BrowserGym, AgentLab propose des mécanismes avancés de gestion des expérimentations, incluant l'exécution parallèle d'expériences, la gestion automatique des erreurs et des dépendances entre tâches, ainsi qu'un suivi détaillé des versions logicielles utilisées pour assurer la reproductibilité. Il offre également des outils interactifs pour l'analyse détaillée des traces, permettant une compréhension approfondie du comportement et des décisions des agents.

L'évaluation expérimentale approfondie, menée sur plusieurs benchmarks représentatifs tels que MiniWoB++, WebArena et WorkArena++, démontre la capacité d'AgentLab à faciliter la comparaison rigoureuse des modèles. Les résultats obtenus soulignent à la fois les performances prometteuses des meilleurs modèles actuels tels que Claude 3.5 Sonnet, mais aussi leurs limites persistantes face à certaines tâches complexes.

Enfin, ce travail propose des perspectives d'amélioration future incluant l'automatisation avancée de l'analyse d'erreurs, la gestion optimisée des ressources computationnelles et l'élargissement de l'écosystème à d'autres domaines d'application. AgentLab constitue ainsi une étape concrète vers la standardisation et la reproductibilité dans l'évaluation des agents web, facilitant une recherche plus transparente, collaborative et orientée vers l'impact.

ABSTRACT

Web agents based on Large Language Models (LLM) are emerging as a promising technology for automating complex interactions with web interfaces. Despite their significant potential, rigorous evaluation of these agents remains challenging due to fragmented existing benchmarks, instability in web environments, and the absence of clearly established methodological standards.

This master thesis introduces AgentLab, a modular and extensible experimental framework developed to address these methodological challenges. Integrated into the BrowserGym ecosystem, AgentLab offers advanced mechanisms for experiment management, including parallel experiment execution, automatic error handling, task dependency management, and detailed version tracking to ensure reproducibility. Additionally, it provides interactive tools for detailed trace analysis, enabling deep insights into agent behavior and decision-making.

A comprehensive experimental evaluation conducted on several representative benchmarks, such as MiniWoB++, WebArena, and WorkArena++, demonstrates AgentLab's capability to facilitate rigorous model comparison. The results highlight both the promising performance of current leading models such as Claude 3.5 Sonnet, and their persistent limitations when faced with certain complex tasks.

Finally, this work proposes future avenues for improvement, including advanced automation of error analysis, optimized computational resource management, and extending the ecosystem to other application domains. AgentLab thus represents a concrete step towards standardization and reproducibility in web agent evaluation, fostering more transparent, collaborative, and impactful research.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

LLM	Grands modèles de language (*Large Language Models*)
RL	Apprentissage par renforcement (*Reinforcement Learning*)
RNN	Réseau de neurones récurrent (*Recurrent neural network*)
LSTM	*Long-Short Term Memory*
GPT	*Generative Pre-trained Transformer*
RLHF	*Reinforcement Learning from Human Feedback*
DPO	*Direct Preference Optimization*
GRPO	*Group Relative Policy Optimization*
API	Interface de programmation d'application (*Application Programming Interface*)
CoT	*Chain-of-Thoughts*
ToT	*Tree-of-Thoughts*
RCI	*Reflect, Criticize, Improve*
SPA	*SeePlanAct*
RAG	*Retrieval Augmented Generation*
DOM	*Document Object Model*

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

1.1 Éléments de la problématique

L'émergence des modèles de langage de grande taille (*Large Language Models*, LLM) a ouvert la voie à une nouvelle génération d'agents intelligents capables d'interagir avec des interfaces web complexes. Ces agents web, en exécutant des instructions de haut niveau à travers un navigateur, permettent d'automatiser de nombreuses tâches quotidiennes ou professionnelles, allant de la simple recherche d'information à la gestion de processus complexes dans des applications d'entreprise. En s'appuyant sur les capacités de raisonnement, de lecture et d'interaction des LLMs, ces agents sont capables d'explorer dynamiquement des pages web, de remplir des formulaires, d'extraire des données, de prendre des décisions conditionnelles et même de dialoguer avec l'utilisateur pour clarifier ses intentions.

Ils portent une promesse technologique majeure : déléguer des tâches numériques fastidieuses à des assistants automatisés, capables de naviguer, cliquer, extraire, et formuler des réponses à partir d'interfaces web standards. Cette automatisation a le potentiel de transformer en profondeur le rapport aux outils numériques, en soulageant les utilisateurs de certaines charges cognitives et temporelles. Au-delà du gain de productivité, ces technologies visent à offrir une assistance précieuse à des populations spécifiques comme les personnes âgées ou en situation de handicap, notamment les non-voyants, pour qui le web reste parfois difficilement accessible. Elles ouvrent également la perspective d'une accessibilité accrue à certains services dématérialisés, en particulier dans les contextes administratifs ou médicaux. En entreprise, les agents web pourraient progressivement remplacer certaines activités répétitives de type "copier-coller", extraction de rapports ou saisie de données, libérant ainsi du temps pour des tâches à plus forte valeur ajoutée.

Pourtant, malgré l'intérêt croissant de la communauté scientifique et l'accélération des publications sur le sujet, la recherche sur les agents web souffre encore d'une forte fragmentation. Chaque benchmark propose son propre format de données, ses outils d'exécution, ses scripts d'entraînement, son interface et ses métriques. Cette diversité, bien que reflet d'une activité créative et innovante, freine la comparaison rigoureuse des méthodes, multiplie les efforts d'intégration, et rend difficile la construction de standards communs. Elle contribue aussi à une grande hétérogénéité dans les pratiques d'évaluation, où les résultats publiés sont rarement directement comparables.

Cette fragmentation est aggravée par la nature particulière du web. Contrairement à des

environnements fermés et déterministes, les interfaces web sont dynamiques, souvent volatiles, et susceptibles d'être modifiées à tout moment. Les sites changent fréquemment de structure, d'apparence, ou de logique de navigation, ce qui rend l'évaluation fragile et sensible à des facteurs extérieurs non contrôlés. Un agent performant aujourd'hui sur une tâche donnée peut échouer par la suite sans que son architecture ou ses stratégies n'aient changé. Par ailleurs, les interactions avec des LLMs via des API commerciales ajoutent un degré supplémentaire d'instabilité, puisque les modèles peuvent évoluer sans préavis.

Dans ce contexte, la standardisation des pratiques, l'amélioration de la robustesse expérimentale, et la mise en place d'outils facilitant la reproductibilité apparaissent comme des besoins fondamentaux. Il devient urgent de proposer des infrastructures permettant de définir des méthodologies communes, de centraliser les benchmarks, d'exécuter les agents de façon contrôlée, et de documenter finement les conditions d'évaluation. C'est dans ce cadre que s'inscrit le présent travail.

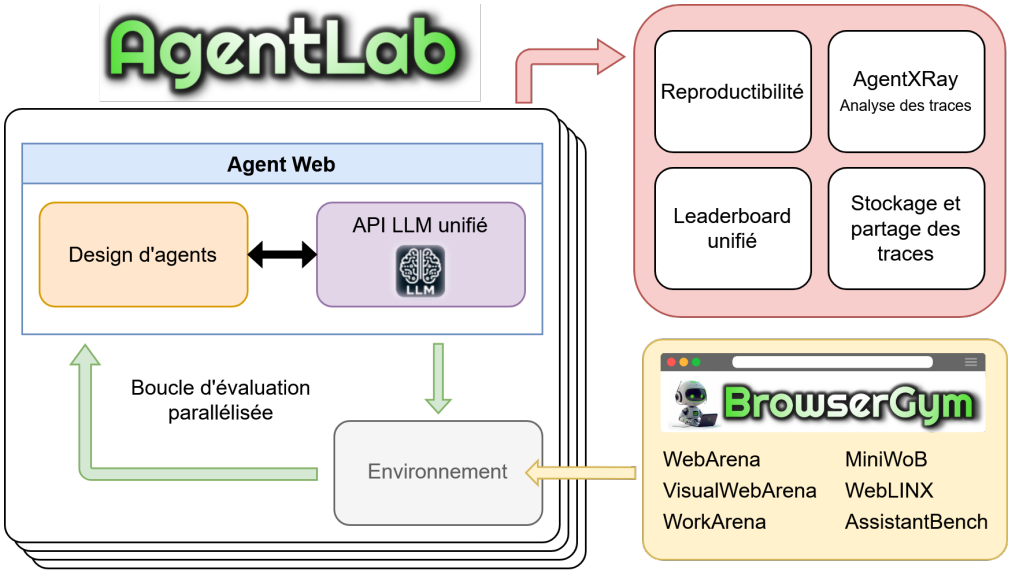


FIGURE 1.1 Schéma global de l'écosystème, incluant AgentLab, BrowserGym et les benchmarks d'évaluation des agents web. AgentLab sert de support pour la création et l'évaluation des agents.

1.2 Objectif de la recherche

Ce travail présente AgentLab, un cadriciel modulaire et extensible conçu pour structurer la recherche sur les agents web. Dans un contexte de fragmentation des benchmarks, d'instabilité des environnements et de difficultés à comparer les méthodes existantes, AgentLab propose

une infrastructure commune pour le développement, l'exécution et l'analyse d'agents. Il vise à favoriser une recherche plus reproductible, plus transparente, et plus accessible, en s'appuyant sur des outils simples à utiliser et à personnaliser.

L'un des objectifs principaux d'AgentLab est d'améliorer la reproductibilité des expériences. Pour cela, le framework, illustré par la figure 1.1 fournit un ensemble de mécanismes pour contrôler finement les conditions d'exécution : fixations de germes aléatoires (*seeds*), journalisation détaillée, relance automatique d'épisodes en cas d'erreur, et sauvegarde des versions de code. L'utilisateur peut ainsi relancer une expérience dans un contexte identique, ou comparer plusieurs configurations avec un haut degré de rigueur expérimentale.

Par ailleurs, AgentLab vise à rendre la recherche plus accessible, en mettant à disposition une interface unifiée pour définir des agents, les exécuter sur différents benchmarks, et analyser leur comportement. Il propose également un outil d'inspection interactif, AgentXRay, qui permet d'explorer les décisions prises par l'agent à chaque étape, de visualiser ses observations et raisonnements, et d'identifier les causes d'échec ou de succès. Cette simplicité d'utilisation permet d'abaisser la barrière d'entrée pour les chercheurs, tout en fournissant des outils puissants pour ceux qui souhaitent prototyper rapidement de nouvelles idées.

En synthèse, AgentLab se positionne comme une réponse concrète aux besoins de standardisation et de transparence dans le domaine des agents web. Il facilite les expérimentations à grande échelle, la comparaison de modèles, et l'analyse fine des comportements, tout en encourageant une démarche de recherche ouverte, reproductible et collaborative. La librairie AgentLab disponible sous forme de logiciel libre de droit pour la communauté scientifique.

1.3 Impact

Les travaux présentés dans ce mémoire s'inscrivent dans une dynamique de structuration de la recherche sur les agents intelligents, et plus particulièrement les agents web. En proposant une infrastructure expérimentale unifiée, AgentLab cherche à réduire les barrières à la collaboration scientifique et à encourager des pratiques de recherche plus rigoureuses, reproductibles et ouvertes.

Du point de vue de la communauté, AgentLab facilite la comparaison entre modèles, la reproduction d'expériences existantes, ainsi que le prototypage d'idées nouvelles. Il s'inscrit dans une logique de mutualisation des outils et des bonnes pratiques, à l'image de ce que des plateformes comme HuggingFace [2] ou OpenAI Gym [3] ont permis dans leurs domaines respectifs. En ce sens, il peut contribuer à accélérer le développement de la recherche sur les agents, en permettant à de nouvelles équipes de s'impliquer plus facilement dans le domaine.

Sur un plan sociétal, les agents web eux-mêmes présagent une transformation de la façon dont les humains interagissent avec les systèmes numériques. S'ils deviennent plus accessibles, plus robustes et mieux contrôlés, ils pourraient rendre de nombreux services à des publics variés : aide à la navigation pour les personnes en situation de handicap, assistants personnels pour les tâches administratives, outils d'automatisation pour les professions exposées à la répétition. Toutefois, cette évolution pose aussi des questions éthiques et économiques : risque de perte d'emploi sur certaines tâches, nécessité d'encadrer les actions des agents, et importance de leur transparence et de leur explicabilité.

En somme, les contributions d'AgentLab se situent à la fois sur le plan scientifique, en renforçant les fondations méthodologiques de l'évaluation des agents, et sur le plan sociétal, en soutenant le développement d'agents plus fiables, performants et adaptés aux besoins des utilisateurs.

1.4 Sommaire des contributions

Dans le cadre de ce travail, j'ai participé à la création d'AgentLab. J'ai aussi assuré la maintenance et l'ajout de fonctionnalités à AgentLab et BrowserGym. Parmi ces fonctionnalités, on compte :

- L'unification des API LLM et l'implémentation d'un mécanisme de suivi des coûts.
- Des mécanismes de relance d'expérimentations en cas d'erreurs.
- Différents outils de visualisation dans AgentXRay.
- Plusieurs outils de création d'agent, avec notamment des outils pour simplifier la génération de prompts.

J'ai aussi eu l'occasion d'implémenter les différents agents présentés dans le texte, `WrapperAgent` et `WebDreamer`, ainsi que le pipeline d'analyse automatique d'erreur.

Finalement, j'ai géré et exécuté l'intégralité des expériences présentées dans le texte.

Les contributions présentées dans ce mémoire ont fait l'objet d'une publication scientifique acceptée à la revue *Transactions on Machine Learning Research* (TMLR) [4], soulignant ainsi leur apport reconnu par la communauté académique. De plus, le travail effectué a permis la réalisation du benchmark WorkArena++, qui a fait l'objet d'une publication à la conférence *Neural Information Processing Systems* (NeurIPS) [1]. De plus, le projet AgentLab dépasse 300 étoiles sur GitHub, soulignant l'adoption de la librairie par une part de la communauté.

CHAPITRE 2 REVUE DE LITTÉRATURE

2.1 Apprentissage des modèles de langage

Les agents web modernes s'appuient sur de grands modèles de langage (LLM), pré-entraînés sur de vastes corpus textuels et adaptés à diverses tâches. Pour mieux comprendre leurs fondements, nous rappelons brièvement les principales approches de l'apprentissage automatique et leur lien avec le développement de ces modèles.

2.1.1 Apprentissage supervisé

L'apprentissage supervisé repose sur l'entraînement d'un modèle à partir de données annotées, sous la forme de couples (entrée, sortie). L'objectif est de généraliser à de nouvelles données en minimisant une fonction de perte qui mesure l'écart entre les sorties prédites et les sorties attendues. Cette approche, au cœur de la majorité des tâches d'intelligence artificielle, est présentée en détail par Hastie et al. [5].

Dans le cas des LLMs, la tâche supervisée typique est la prédiction du mot suivant. Le modèle est entraîné à partir de textes bruts, considérés comme une séquence de jetons (*tokens*) $(x_1, x_2, ..., x_T)$, où il doit apprendre à prédire x_{t+1} à partir des jetons précédents. Chaque séquence d'entrée devient ainsi un exemple supervisé implicite. Ce cadre de pré-entraînement auto-supervisé s'étend à d'autres variantes, comme la prédiction de parties masquées dans un texte.

Le succès de l'apprentissage supervisé repose sur la disponibilité de données massives et sur des architectures capables de modéliser efficacement les dépendances entre jetons, comme les *Transformers* [6]. Ce pré-entraînement permet aux modèles de développer des représentations riches du langage, qui peuvent ensuite être réutilisées pour des tâches variées, y compris l'interaction dans des environnements comme le web.

2.1.2 Apprentissage par renforcement

L'apprentissage par renforcement (*Reinforcement Learning*, RL), tel que formalisé par Sutton et Barto [7], modélise un agent qui apprend à interagir avec un environnement en vue de maximiser une récompense cumulative. Contrairement à l'apprentissage supervisé, le retour n'est pas donné directement sous forme d'étiquette, mais sous forme de signal de récompense, parfois différé.

Le cadre classique est celui du processus de décision markovien, où à chaque étape :

- l'agent observe un état s_t de l'environnement,
- choisit une action a_t selon une politique $\pi(a|s)$,
- et reçoit une récompense r_t tout en passant à un nouvel état s_{t+1}.

L'objectif de l'agent est de maximiser la somme pondérée des récompenses futures, souvent notée $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$, où γ est un facteur d'atténuation des récompenses futures. Cette approche est particulièrement pertinente dans le cadre d'agents autonomes évoluant dans des environnements séquentiels. Les agents doivent y prendre des décisions sur plusieurs étapes, en apprenant à maximiser leur performance globale à travers l'interaction avec des environnements complexes. Bien que l'apprentissage par renforcement soit moins directement impliqué dans l'entraînement initial des LLMs, il constitue une base conceptuelle essentielle dès lors qu'on cherche à utiliser ces modèles dans des environnements dynamiques, comme les navigateurs web, où l'agent agit en boucle perception-action.

2.1.3 Apprentissage profond et architectures neuronales

Les LLMs reposent sur l'apprentissage profond, c'est-à-dire l'utilisation de réseaux de neurones comportant de nombreuses couches, entraînés via descente de gradient et rétropropagation. Cette approche, popularisée par des architectures convolutionnelles pour la vision [8], s'est étendue au traitement du langage avec des modèles séquentiels comme les réseaux récurrents (*Recurrent Neural Networks*, RNN) et les LSTM (*Long-Short Term Memory*) [9], capables de capturer des dépendances temporelles. On se réfère ici à Bengio & Courville [10] pour une présentation complète de ces architectures.

Cependant, ces architectures montrent des limitations sur de longues séquences, en particulier des problèmes de disparition ou d'explosion du gradient, et d'une difficulté dans la parallélisation de l'entraînement.

2.1.4 Les Transformers et l'avènement des LLMs

L'introduction des *transformers* [6] a marqué un tournant majeur pour le traitement du langage naturel. Cette architecture élimine la récursivité au profit d'un mécanisme d'auto-attention, qui permet à chaque jeton de pondérer dynamiquement l'ensemble de la séquence. Ce fonctionnement autorise un entraînement entièrement parallèle et améliore la capacité à capturer des relations à longue distance dans le texte.

Les LLMs modernes, tels que GPT [11], Claude [12] ou LLaMA [13], s'appuient sur des architectures de type *transformer decoder-only* [14], dans lesquelles chaque jeton est prédit à

partir des précédents. Ces modèles sont pré-entraînés sur de larges corpus (p.ex., web, livres et code), puis affinés via supervision ou apprentissage par renforcement pour accomplir des tâches spécifiques, comme la génération de texte, la recherche d'information ou l'exécution d'actions dans un navigateur web.

2.2 Loi d'échelle dans le réseaux neuronaux

Depuis les premiers *transformers*, l'évolution des modèles de langage a été marquée par une augmentation massive du nombre de paramètres, des données d'entraînement et des ressources de calcul. Cette croissance suit des lois d'échelle empiriques, mises en évidence par Kaplan et al. [15]. Ces travaux montrent qu'il existe une relation prévisible entre la performance des modèles et trois facteurs clés : la taille du modèle (i.e., le nombre de paramètres), la quantité de données d'entraînement, et le budget de calcul utilisé (en FLOPs). Sous certaines hypothèses, les pertes des modèles diminuent de façon régulière en fonction de ces ressources, suggérant qu'il n'existe pas de plafond fondamental à l'amélioration par simple montée en échelle.

Ces lois d'échelle ont joué un rôle structurant dans le développement des grands modèles comme GPT-3 et GPT-4 [16] ou LLaMA [13], en guidant les arbitrages entre taille du modèle et taille du corpus. Toutefois, les hypothèses initiales ont été remises en question par des travaux ultérieurs, qui ont montré que les grands modèles publiés étaient souvent sous-entraînés, c'est-à-dire trop gros pour la quantité de données effectivement utilisée. Leurs résultats, synthétisés dans le paradigme dit de *Chinchilla* [17], suggèrent qu'un modèle plus petit, mais mieux entraîné (i.e., plus de jetons vus), peut surpasser un modèle plus gros sous-entraîné, à budget constant.

Ces analyses ont des conséquences pratiques importantes : elles permettent non seulement de planifier le développement de nouveaux modèles de façon plus efficace, mais elles donnent aussi des pistes pour l'optimisation énergétique et l'allocation des ressources. Par ailleurs, la montée en échelle des modèles semble également favoriser l'émergence de capacités nouvelles, telles que le raisonnement [18, 19, 20, 21, 22], la capacité à utiliser des outils [22, 23, 24, 25, 26, 27, 28, 29, 30], ou la résolution de tâches complexes en plusieurs étapes [20, 21, 22, 30], qui seront particulièrement importantes dans le contexte des agents autonomes que nous décrivons dans la suite de ce texte.

2.3 Alignement des LLMs

À mesure que les modèles de langage deviennent plus puissants et polyvalents, la question de leur alignement avec les intentions humaines devient centrale. L'alignement vise à orienter les comportements des modèles de façon à ce qu'ils produisent des réponses utiles, sûres, et conformes aux préférences humaines [31]. Ce processus ne se limite pas à des considérations éthiques : il joue également un rôle structurant dans l'émergence de capacités agentiques. En effet, les techniques d'alignement encouragent les modèles à adopter des raisonnements plus structurés, à suivre des instructions complexes, ou à proposer des plans d'action. On peut ainsi considérer l'alignement comme un mécanisme qui sélectionne, affine, et amplifie les capacités latentes des LLM, leur permettant de mieux simuler des comportements orientés vers un but, préparant le terrain pour les agents que nous étudierons dans les sections suivantes.

Une approche couramment utilisée est le *reinforcement learning from human feedback* (RLHF) [32]. Cette méthode consiste à entraîner un modèle à partir de préférences humaines, en le récompensant pour des réponses jugées satisfaisantes et en le pénalisant pour celles qui ne le sont pas. Le processus typique du RLHF comprend trois étapes : un pré-entraînement initial, la formation d'un modèle de récompense basé sur les préférences humaines, et l'optimisation de la politique du modèle via des algorithmes tels que *proximal policy optimization* (PPO) [33]. Bien que le RLHF ait démontré son efficacité, il présente des défis, notamment la sensibilité aux hyperparamètres et la complexité de sa mise en œuvre.

Pour simplifier le processus d'alignement, le *direct preference optimization* (DPO) [34] a été proposé. Contrairement au RLHF, le DPO élimine le besoin d'un modèle de récompense distinct et d'une boucle d'apprentissage par renforcement. Il ajuste directement le modèle principal en fonction des préférences humaines, traitant l'alignement comme un problème d'apprentissage supervisé sur des données de préférence. Cette simplification réduit la complexité computationnelle et facilite l'entraînement, tout en maintenant des performances comparables.

Récemment, le modèle DeepSeek-R1 [35] a introduit une approche novatrice en matière d'alignement. Développé par DeepSeek, ce modèle exploite l'apprentissage par renforcement pur pour améliorer les capacités de raisonnement des grands modèles de langage, sans recourir à des données supervisées. En utilisant *group relative policy optimization* (GRPO) [36], DeepSeek-R1 a démontré des performances comparables à celles des modèles de pointe, tout en réduisant les coûts et le besoin d'annotations humaines.

Ces différentes méthodes illustrent la diversité des stratégies d'alignement des modèles de langage. Le choix de l'approche dépend des ressources disponibles, des objectifs spécifiques

et des contraintes éthiques associées à chaque application.

2.4 Agents et Agents Web

Le concept d'agent autonome couvre une variété de systèmes informatiques capables de percevoir leur environnement et d'y agir de manière indépendante afin d'atteindre des objectifs prédéfinis. Ces agents incluent les agents logiciels, capables d'interagir avec des systèmes informatiques via des API, ainsi que les agents physiques capables d'interagir avec leur environnement via des capteurs et des actionneurs.

Un agent basé sur les LLMs (en anglais *LLM-based agent*) est un algorithme ou un système qui utilise un LLM pour effectuer des tâches de façon autonome. Ces agents tirent parti de la capacité des LLMs à raisonner et à traiter différentes formes de texte, pour en faire des assistants dans diverses applications comme la résolution de problèmes ou l'aide à la clientèle.

Récemment, des agents web basés sur les LLMs ont émergé, tirant parti directement de la nature principalement textuelle du web (par exemple, HTML, AXTree) pour automatiser des interactions via les interfaces utilisateur (UI) conçues pour les humains, telles que remplir des formulaires, naviguer sur des sites d'e-commerce ou effectuer des commandes. Ces agents web offrent l'avantage d'une grande adaptabilité et d'une compatibilité élargie, sans nécessiter le développement d'interfaces spécifiques, ce qui améliore notamment l'accessibilité numérique pour les utilisateurs en situation de handicap.

Afin d'améliorer la performance de ces agents, deux aspects principaux sont généralement abordés dans la recherche actuelle : le raisonnement et l'usage d'outils. Nous approfondirons ces deux dimensions dans les sections suivantes.

2.4.1 Méthodes de Raisonnement

Les agents basés sur des modèles de langage tirent parti de plusieurs stratégies de raisonnement pour améliorer leur efficacité, notamment dans des environnements complexes comme le web. Une approche fondatrice est le *chain-of-thought prompting* (CoT) [18], qui consiste à expliciter les étapes intermédiaires du raisonnement en langage naturel. En fournissant des exemples annotés contenant des chaînes de raisonnement, les LLMs sont capables de mieux structurer leur processus génératif et d'améliorer leurs performances sur des tâches telles que le raisonnement mathématique ou symbolique. Cependant, cette méthode repose sur une génération linéaire et séquentielle, ce qui limite la capacité à explorer plusieurs chemins alternatifs ou à revenir en arrière en cas d'erreur.

Pour surmonter ces limitations, le paradigme *tree of thoughts* (ToT) [19] généralise l'approche CoT en introduisant une structure de raisonnement arborescente. Plutôt que de produire une seule chaîne linéaire, le modèle explore un arbre de pensées, où chaque nœud représente une étape intermédiaire possible. À chaque étape, le modèle peut générer plusieurs pensées candidates, les évaluer, les comparer, et choisir celles qui méritent d'être poursuivies, selon des heuristiques implicites ou explicites. Cette approche a montré des gains significatifs sur des tâches complexes comme les mathématiques ou la rédaction créative, tout en restant compatible avec des modèles pré-entraînés sans phase de fine-tuning.

En parallèle, la méthode *ReAct* [20] propose une extension complémentaire à CoT, non pas pour en corriger les limites structurelles, mais pour l'intégrer dans un cadre d'agent interactif. ReAct combine des pensées explicites avec des actions concrètes effectuées dans un environnement, ainsi que des observations issues de ces actions. Cette boucle perception-réflexion-action permet à l'agent d'interroger dynamiquement son environnement, par exemple via un moteur de recherche ou l'inspection d'une interface web, de réviser ses raisonnements en fonction des observations, et d'adapter sa séquence d'interactions.

Une autre stratégie complémentaire est le *RCI (Reflect, Criticize, Improve)* [21], qui introduit une boucle de rétroaction explicite. L'agent génère une réponse initiale, puis l'évalue lui-même, identifie les erreurs ou incohérences, et propose une amélioration. Cette méthode exploite les capacités d'auto-critique des LLMs pour affiner leur production.

Enfin, des travaux comme *SeePlanAct* (SPA) [22], introduit dans le benchmark *Assistant-Bench*, montrent l'intérêt de la planification explicite dans les agents web. SPA enrichit un agent interactif avec un module de planification et une mémoire, permettant de formuler un plan global, de le réviser dynamiquement, et de stocker les informations extraites au fil des interactions. Cette capacité à structurer l'action dans le temps, combinée avec des raisonnements guidés par le contexte, améliore sensiblement la performance sur des tâches longues, réalistes et multi-étape, où les agents traditionnels échouent.

2.4.2 Usage d'Outils

Les capacités des modèles de langage peuvent être considérablement étendues par l'usage d'outils externes. Cette approche permet de dépasser les limites des LLMs (connaissance figée, fenêtre de contexte limité, incapacité à exécuter du code) en les dotant d'un accès actif à des sources d'information ou à des environnements d'interaction. Ces outils jouent un rôle central dans la construction d'agents capables d'agir de façon autonome, notamment dans des contextes web, logiciels ou scientifiques.

Un premier type d'outil couramment intégré dans les agents est un moteur de recherche, via l'approche *Retrieval-Augmented Generation* (RAG). Celle-ci consiste à doter le modèle d'un accès à une base de documents ou à des résultats issus du web, qu'il peut interroger dynamiquement. Introduite dans des travaux comme REALM [24] ou RAG [23], cette technique permet aux LLMs de compléter leur raisonnement par des passages pertinents extraits d'un corpus externe. RAG est aujourd'hui utilisé dans de nombreux agents interactifs, notamment lorsqu'ils s'appuient sur des corpus de connaissances pour répondre à des questions complexes.

Au-delà de la simple recherche, certains agents sont capables d'interagir directement avec un navigateur web. C'est le cas de WebGPT [25], Mind2Web [26], WebVoyager [27] ou encore SPA [22]. Ces agents peuvent lire les pages HTML, cliquer, remplir des formulaires ou naviguer à travers des interfaces humaines. Cela leur permet d'effectuer des tâches réalistes et ouvertes, comme réserver un vol ou consulter des avis. Toutefois, ces environnements restent difficiles à maîtriser de manière fiable : les agents peuvent mal interpréter les éléments de la page, prendre des décisions erronées ou échouer à coordonner leurs actions à long terme.

Dans un autre registre, certains agents intègrent l'accès à des API, leur permettant d'exécuter du code, d'effectuer des calculs, de manipuler des fichiers ou de produire des visualisations. Cette capacité est particulièrement utile pour des tâches scientifiques, analytiques ou de transformation de données. Des systèmes comme le Code Interpreter d'OpenAI, ou des travaux comme Toolformer [28] et AnyTool [29], illustrent bien cette direction. Les LLM y apprennent à déclencher automatiquement des appels à des fonctions spécialisées, pour déléguer certaines opérations à des outils externes.

Une tendance récente vise à doter les agents d'une capacité à interagir avec un environnement informatique complet. OSWorld [30], par exemple, simule un bureau virtuel avec terminal, navigateur, explorateur de fichiers ou éditeur de texte. Les agents doivent y exécuter des instructions complexes dans un contexte réaliste, proche d'un ordinateur personnel. Cela ouvre la voie à des agents plus généralistes, capables d'enchaîner des actions variées dans un cadre multi-outils et persistant.

2.5 Présentation des benchmarks

L'émergence récente des agents web, qui combinent les capacités avancées des modèles de langage et de vision pour effectuer des interactions complexes sur des interfaces web, soulève naturellement la question de leur évaluation rigoureuse. En effet, ces agents promettent d'automatiser efficacement des tâches quotidiennes variées, allant de la simple navigation

sur des sites à la réalisation de tâches plus sophistiquées nécessitant des interactions contextuelles approfondies avec les éléments d'interfaces graphiques utilisés par les humains. Pour mesurer les capacités réelles de ces agents et assurer leur amélioration continue, une multitude de benchmarks ont été proposés. Ces benchmarks cherchent à refléter la diversité et la complexité des scénarios d'utilisation du web, tout en permettant une évaluation systématique et reproductible des performances des différents modèles d'agents web proposés par la communauté scientifique.

2.5.1 MiniWoB

MiniWoB++ [37] est une version étendue du benchmark MiniWoB (Minimal World of Bits), conçue pour évaluer la capacité des agents web à effectuer diverses tâches interactives dans un navigateur. MiniWoB++ propose un ensemble d'environnements web synthétiques permettant d'évaluer des agents d'apprentissage par renforcement dans des tâches interactives. Parmi les principales caractéristiques du benchmark, la diversité des tâches et les variabilités stochastiques proposées font de MiniWoB++ un point d'entrée clé dans le développement et l'étude des agents web. La figure 2.1a montre l'une des tâches de MiniWoB.

MiniWoB++ se distingue par sa collection de tâches couvrant un large éventail d'interactions utilisateur typiques sur le web, telles que le remplissage de formulaires, le clic sur des boutons et la saisie de texte. Ces tâches sont conçues pour être représentatives des interactions que l'on retrouve couramment dans les interfaces web modernes, tout en introduisant des défis spécifiques pour les agents automatique.

Un autre atout majeur de MiniWoB++ réside dans son cadre expérimental contrôlé, où chaque tâche est implémentée dans un environnement HTML minimaliste. Cette propriété élimine tout facteur externe pouvant influer sur les performances d'un agent, tout en conservant des structures et une logique proche des interfaces utilisateurs standards. De plus, la stochasticité des environnements force les agents à développer des stratégies robustes et généralisables.

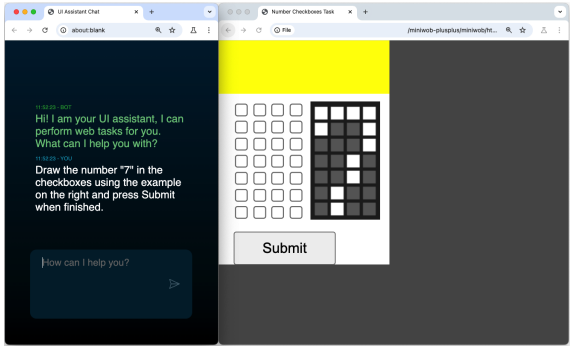
Le benchmark propose 125 tâches différentes. Ces tâches sont courtes, et suivent un schéma de récompense binaire, où l'agent reçoit une récompense de 1 en cas de succès de la tâche, et de 0 en cas d'échec. Ces récompenses clairsemées (*sparse*) forcent les agents à explorer les environnements en profondeur avant de recevoir un signal d'apprentissage, rendant les méthodes d'apprentissage par renforcement plus difficiles à appliquer.

2.5.2 WebArena

WebArena [38] est un benchmark conçu pour l'évaluation des agents autonomes dans des environnements web réalistes et interactifs. Contrairement aux approches plus synthétiques comme MiniWoB++, WebArena propose des sites web fonctionnels inspirés de domaines courants tels que le commerce électronique, les forums de discussion, le développement collaboratif et la gestion de contenu. L'objectif est d'évaluer la capacité des agents à interagir avec des interfaces web variées et à exécuter des tâches complexes en suivant des instructions formulées en langage naturel. L'une des tâches de WebArena est visible dans la figure 2.1b.

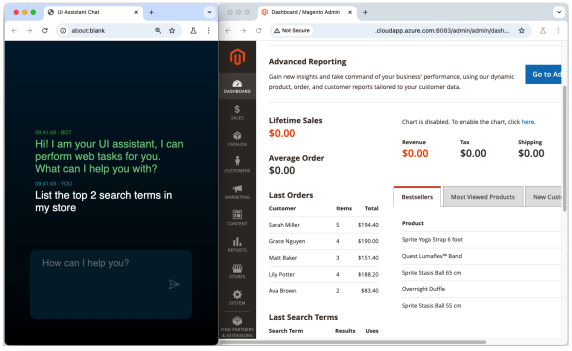
WebArena se distingue par la diversité et la complexité des tâches proposées. Le benchmark comprend 812 tâches, couvrant un large éventail d'interactions, telles que la recherche d'informations, la navigation sur des plateformes en ligne et l'exécution d'actions spécifiques sur des interfaces utilisateur. Contrairement à certains benchmarks basés sur des comparaisons strictes avec des séquences d'actions préétablies, WebArena évalue les agents sur leur capacité à atteindre un objectif fonctionnel, permettant ainsi une plus grande flexibilité dans les stratégies adoptées.

L'état de l'art actuel atteint un taux de succès de 61,7%, ce qui reflète des avancées significatives par rapport aux premières évaluations, où les meilleurs agents basés sur GPT-4 obtenaient seulement 14,41%. Toutefois, ce taux reste encore largement inférieur aux 78,24% de réussite des humains, mettant en évidence les défis persistants liés à la compréhension et à l'exécution de tâches complexes sur le web. WebArena constitue ainsi un benchmark clé pour mesurer les progrès en intelligence artificielle appliquée à l'interaction web.



`gym.make("browsergym/miniwob.number-checkboxes")`

(a) Exemple de tâche MiniWoB.



`gym.make("browsergym/webarena.399")`

(b) Exemple de tâche WebArena

FIGURE 2.1 Deux tâches, respectivement de MiniWoB et de WebArena, accessibles via BrowserGym.

2.5.3 WorkArena

WorkArena [39] est un benchmark novateur conçu pour évaluer les capacités des agents web dans l'accomplissement de tâches réalistes effectuées régulièrement dans un contexte d'entreprise. Développé sur la plateforme ServiceNow, WorkArena propose 33 tâches distinctes, regroupées en différents niveaux de difficulté, visant à tester rigoureusement les compétences de planification, de résolution de problèmes, de raisonnement logique et arithmétique, ainsi que de récupération d'information des agents web. La figure 2.2 illustre l'une des tâches du benchmark.

Contrairement à MiniWoB++ qui se focalise sur des interactions simples et synthétiques, WorkArena met l'accent sur des scénarios de travail réalistes nécessitant des stratégies complexes et une compréhension approfondie du contexte professionnel. Tandis que WebArena propose des interactions diversifiées et réalistes en simulant des sites web fonctionnels issus de catégories telles que les forums sociaux, le commerce en ligne ou la gestion de contenu, WorkArena se concentre spécifiquement sur les environnements professionnels typiques via la plateforme ServiceNow. Chaque tâche est intégrée à un environnement contrôlé mais réaliste, dans lequel les agents doivent interagir avec des interfaces utilisateur représentant fidèlement celles rencontrées en milieu professionnel. L'environnement offre ainsi une combinaison de tâches atomiques et composites, nécessitant de gérer simultanément plusieurs étapes intermédiaires afin d'atteindre les objectifs finaux.

WorkArena est développé nativement dans l'écosystème BrowserGym, une plateforme unifiée facilitant l'évaluation standardisée des agents web. Cette intégration permet une gestion cohérente des observations et des actions des agents. La validation des tâches est automatisée et rigoureuse, vérifiant systématiquement l'état de la base de données et de l'interface utilisateur pour déterminer la réussite ou l'échec des agents, assurant ainsi des résultats fiables et cohérents.

2.5.4 WebLINX

WebLINX [40] est un benchmark qui introduit le concept novateur de navigation conversationnelle sur des sites web réels via un dialogue à plusieurs tours. Constitué de 100 000 interactions réparties sur 2 300 démonstrations effectuées par des experts, ce benchmark couvre une large variété de tâches impliquant plus de 150 sites web authentiques. L'objectif de WebLINX est d'évaluer les capacités des agents à suivre des instructions utilisateur complexes et évolutives, tout en naviguant dans un environnement dynamique nécessitant une compréhension contextuelle approfondie des éléments web. Le benchmark met parti-

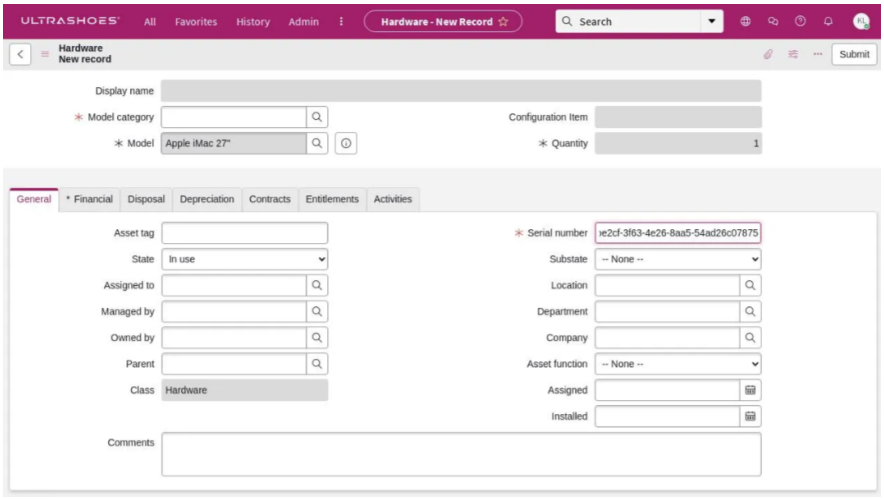


FIGURE 2.2 Illustration d'une tâche de complétion de formulaire dans WorkArena.

culièrement l'accent sur l'importance du dialogue interactif, en examinant la capacité des agents à interagir efficacement en temps réel avec les utilisateurs à travers plusieurs tours de conversation.

2.5.5 AssistantBench

AssistantBench [22] propose un ensemble de 214 tâches réalistes visant spécifiquement à évaluer les capacités des agents web à naviguer sur plusieurs sites pour résoudre des problèmes nécessitant une recherche et une synthèse d'informations. Ce benchmark cible des tâches qui sont naturellement chronophages pour les utilisateurs humains, telles que la surveillance des marchés immobiliers ou la recherche de commerces locaux pertinents. AssistantBench met en évidence les limites actuelles des modèles linguistiques et des agents augmentés par récupération d'informations, démontrant que même les systèmes les plus avancés n'atteignent pas un taux de réussite supérieur à 26 points.

2.5.6 SafeArena

SafeArena [41] est le premier benchmark conçu spécifiquement pour évaluer la sécurité des agents web autonomes face à des utilisations malveillantes délibérées. Composé de 250 tâches sûres et 250 tâches nuisibles, réparties sur cinq catégories de nuisances potentielles (désinformation, activité illégale, harcèlement, cybercriminalité et biais sociaux) SafeArena vise à évaluer systématiquement le comportement des agents face à des requêtes malveillantes réalistes. Une étude préliminaire révèle que même les agents basés sur des modèles avec des

pare-feux efficaces comme GPT-4o peuvent exécuter un pourcentage alarmant de requêtes à motivations malveillantes, soulignant l'importance cruciale d'aligner spécifiquement les procédures de sécurité aux tâches web.

2.6 Présentation de BrowserGym

BrowserGym émerge comme une réponse à un défi majeur identifié dans la recherche sur les agents web automatisés : la fragmentation des benchmarks et des interfaces d'interaction avec le web. Alors que le domaine connaît une rapide expansion, porté par l'apparition constante de nombreux modèles sophistiqués et de benchmarks innovants, il devient crucial de disposer d'une infrastructure cohérente et robuste pour évaluer et comparer ces agents de manière fiable. Cette fragmentation initiale limite considérablement les avancées potentielles en empêchant une analyse comparative rigoureuse et en ralentissant le développement de méthodologies communes.

Dans ce contexte, BrowserGym propose une interface unifiée facilitant non seulement l'intégration efficace des benchmarks existants, mais aussi la création rapide et simplifiée de nouveaux benchmarks adaptés à des contextes variés. Cette plateforme se distingue par son architecture flexible, permettant aux chercheurs d'ajouter aisément de nouvelles tâches et scénarios pour étendre continuellement son champ d'application. BrowserGym offre également une structure claire pour la définition des observations et des actions possibles, facilitant ainsi l'interaction des agents avec les environnements web.

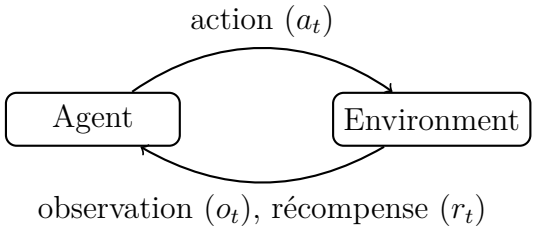
Cette section présente en détail les caractéristiques techniques et conceptuelles de BrowserGym, explorant ses fonctionnalités clés, et ses apports méthodologiques en matière de standardisation.

2.6.1 Le web comme un environnement Gym

BrowserGym adopte une approche directement inspirée de l'apprentissage par renforcement, en modélisant les interactions des agents web comme un processus de décision markovien partiellement observable. Dans ce cadre, chaque action réalisée par l'agent entraîne une nouvelle observation de l'environnement, composée à la fois d'informations textuelles issues du navigateur et d'un dialogue interactif via un interface de discussion, ainsi qu'une récompense définissant la qualité de l'action précédente. Cette boucle d'interaction continue, illustrée par la figure 2.3, permet à l'agent de progresser au travers du web pour accomplir son objectif. BrowserGym implémente l'interface standard OpenAI Gym [3], ce qui facilite grandement l'intégration et l'expérimentation avec des méthodes RL existantes, et est accessible en Py-

thon via l'interface Gymnasium [42]. Techniquement, BrowserGym repose sur le navigateur Chromium et utilise la bibliothèque Playwright pour automatiser de façon précise et fiable les interactions avec les pages web.

En plus d'interactions classiques entre l'agent et le web, BrowserGym joint à chaque environnement un clavardage, dans lequel l'agent peut communiquer directement avec l'utilisateur ou recevoir des instructions supplémentaires. Ce clavardage permet par exemple à l'utilisateur d'offrir à l'agent une meilleure compréhension du contexte et de ses attentes. Le clavardage peut aussi servir à l'agent pour compléter certaines tâches, par exemple de recherche d'information. Il suffit alors à l'agent d'inscrire l'information trouvée dans le clavardage pour soumettre une réponse.




```python
import gymnasium as gym

# création de l'environnement
env = gym.make("browsergym/miniwob.choose-list")

# boucle d'évaluation
obs, info = env.reset()
while True:
  action = ... # l'agent choisit une action
  obs, reward, terminated, truncated, info =
  ↪  env.step(action)
  if terminated or truncated: break
```

FIGURE 2.3 Modèle théorique de la boucle d'évaluation BrowserGym (gauche), Implémentation minimale de la boucle (droite)

2.6.2 L'espace des observations

Les observations dans BrowserGym jouent un rôle central dans l'apprentissage et l'efficacité des agents web. Chaque observation fournie par l'environnement BrowserGym est structurée pour offrir aux agents un ensemble riche et diversifié d'informations pertinentes sur l'état actuel de l'environnement web.

- L'observation inclut une description structurée de la page, qui comprend notamment le modèle d'objets de document (*Document Object Model*, DOM) brut et l'arbre d'accessibilité (AXTree). Le DOM permet aux agents d'accéder au contenu textuel et aux attributs structurels, tandis que l'AXTree fournit une abstraction améliorant l'accessibilité, indiquant les éléments interactifs et leurs relations hiérarchiques.
- BrowserGym fournit des propriétés supplémentaires (`extra_element_properties`) pour les éléments HTML, telles que des identifiants uniques, appelés `bid`, et les coordonnées précises, facilitant ainsi les interactions ciblées et précises des agents avec les éléments

des pages web. BrowserGym inclut aussi les éléments nécessaire à la construction du *Set-of-Marks* de l'observation.

- Chaque observation comprend également une capture d'écran complète de la page actuelle, permettant aux agents utilisant des modèles de vision-langage de mieux interpréter visuellement les pages et contextualiser leurs actions.
- BrowserGym renseigne les agents sur l'état actuel des onglets ouverts, en indiquant les titres et les URLs des pages disponibles, ce qui est particulièrement utile lors de tâches nécessitant une navigation complexe à travers plusieurs onglets.
- Les observations incluent l'objectif et les messages de clavardage, assurant que les agents peuvent suivre précisément les instructions utilisateur ou gérer des dialogues interactifs tout au long de leurs tâches.
- Les agents reçoivent un retour d'erreur immédiat après chaque action, leur permettant de rapidement identifier et corriger leurs erreurs, et facilitant ainsi l'apprentissage et l'adaptation dynamique dans l'environnement.

La figure 2.4 montre les différents éléments disponibles dans les observations proposées par BrowserGym.

```
'chat_messages':            # Intéractions entre l'agent et l'utilisateur
'goal':                     # Texte d'écrivant la tâche à accomplir
'goal_object':              # Objet au format OpenAI avec la tâche à accomplir
'open_pages_urls':          # URL des onglets ouverts
'open_pages_titles':        # Noms des onglets ouverts
'active_page_index':        # Onglet actif
'url':                      # URL de l'onglet actif
'screenshot':               # Capture d'écran de la page active
'dom_object':               # Document Object Model
'axtree_object':            # Accessibility tree
'extra_element_properties': # Informations sur les éléments de la page
'focused_element_bid':      # Elément actuellement sélectionné
'last_action':              # Dernière action executée
'last_action_error':        # Message d'erreur de la dernière action exécutée
'elapsed_time':             # Temps écoulé depuis le début de l'épisode
```

FIGURE 2.4 Inventaire des informations disponible dans les observations BrowserGym

Cette diversité d'informations intégrées assure que les agents disposent de toutes les ressources nécessaires pour exécuter efficacement leurs tâches et facilite le développement d'agents web robustes et performants.

2.6.3 L'espace des actions

Dans BrowserGym, les actions réalisées par les agents sont définies à travers un système structuré et modulable, appelé `action_mapping`. Ce mécanisme permet aux utilisateurs de BrowserGym de convertir les actions spécifiées par l'agent en commandes exécutables de

manière sécurisée et cohérente au sein du navigateur. En effet, l'utilisation d'action prédéfinie permet à BrowserGym d'avoir un contrôle sur leur exécution et d'assurer une exécution correcte de celles-ci.

Plus précisément, l'`action_mapping` est une fonction qui interprète les instructions textes fournies par l'agent sous forme de code Python/Playwright exécutable, tout en garantissant que seules des actions prédéfinies et autorisées soient effectuées. Cette fonctionnalité essentielle contribue à la sécurité et à la stabilité des interactions web automatisées.

BrowserGym propose par défaut des `HighLevelActionSet`, ensembles d'actions de haut niveau prédéfinis pour faciliter les interactions communes avec les pages web. Ces actions incluent, entre autres, le clic sur des éléments spécifiques, la saisie de texte dans les champs appropriés, la gestion des onglets du navigateur, et la navigation entre différentes pages web. L'ensemble des actions disponibles est clairement défini et structuré, permettant ainsi aux agents de planifier et d'exécuter efficacement leurs interactions.

Cette approche structurée et flexible des actions, grâce à l'`action_mapping` et à l'ensemble prédéfini d'actions de haut niveau, simplifie grandement le développement d'agents web robustes et facilite leur intégration dans diverses tâches et scénarios d'utilisation.

2.6.4 Intégration de nouvelles tâches

BrowserGym simplifie considérablement la création de nouvelles tâches web personnalisées, permettant ainsi aux chercheurs et praticiens de définir facilement leurs propres scénarios expérimentaux. Pour cela, la plateforme requiert uniquement l'implémentation de deux méthodes principales : `setup()` et `validate()`.

La méthode `setup()` est responsable d'initialiser l'environnement web à partir d'une page vierge. Typiquement, cette méthode implique de naviguer vers l'URL spécifique de départ de la tâche, et éventuellement de préparer l'environnement en réalisant des actions telles que des connexions à des systèmes ou des entrées de données initiales. À l'issue de cette étape, la méthode renvoie l'objectif que l'agent devra accomplir. Cet objectif peut être spécifié sous la forme d'une instruction textuelle simple ou sous la forme d'un dialogue interactif, permettant ainsi une flexibilité importante dans la définition des scénarios d'évaluation. Ce dialogue peut par exemple inclure des images de référence pour la tâche à accomplir.

La méthode `validate()` intervient après chaque action réalisée par l'agent. Son rôle est d'évaluer si l'objectif défini précédemment a été atteint. Pour cela, elle accède à l'état courant du navigateur ainsi qu'à l'historique des messages échangés dans le clavardage. Cette méthode vérifie généralement des critères spécifiques tels que la présence de données particulières sur

la page ou la conformité d'un message donné. Elle retourne alors une récompense numérique pour l'agent, un indicateur booléen signalant si la tâche est terminée (`done`), ainsi qu'un message optionnel à destination de l'agent, afin de simuler des retours d'utilisateur réalistes.

Ce mécanisme simple et clair de création de tâches offre une grande facilité d'utilisation et une flexibilité maximale, permettant ainsi une variété étendue d'applications et de scénarios expérimentaux.

2.6.5 Unification des benchmarks pour agents web

BrowserGym se distingue particulièrement par sa capacité à intégrer de multiples benchmarks existants, contribuant ainsi significativement à la réduction des silos dans la recherche sur les agents web automatisés. En réunissant sous une interface unifiée des benchmarks variés tels que MiniWoB, WebArena, VisualWebArena, WorkArena (L1, L2, L3), AssistantBench et WebLINX, BrowserGym permet aux chercheurs de couvrir une grande variété de tâches tout en maintenant une structure d'évaluation cohérente.

Cette intégration transparente accélère considérablement la recherche en simplifiant la préparation du backend et en réduisant le bruit lié aux divergences d'implémentations ou de méthodologies entre benchmarks. Les benchmarks WebArena et VisualWebArena nécessitent entre chaque évaluation d'un agent la réinitialisation des instances utilisées pour héberger le serveur. Cette réinitialisation est prise en charge par BrowserGym.

Chaque benchmark intégré est accompagné de métadonnées détaillées, facilitant ainsi la sélection des tâches appropriées pour chaque expérience. Ces métadonnées permettent par exemple sur WorkArena de trier les tâches par type de compétences, de la navigation à la recherche d'information.

BrowserGym fournit également des paramètres d'évaluation recommandés pour chaque benchmark, garantissant que les agents puissent être évalués de manière rigoureuse et comparable. Typiquement, chaque benchmark est livré avec une liste d'actions par défaut, nécessaire à la réalisation des tâches. WorkArena L2 nécessite par exemple l'action `report_infeasible`. Cette standardisation réduit la complexité inhérente à l'intégration individuelle de chaque benchmark, permettant aux chercheurs de concentrer leurs efforts directement sur l'amélioration et l'analyse approfondie des agents.

2.7 Limitations des Agents Web

Malgré les progrès récents dans le développement d'agents basés sur les LLMs, plusieurs limitations persistent. Tout d'abord, les observations extraites du web, comme le DOM ou l'AXTree, sont souvent trop volumineuses pour être directement intégrées dans le contexte des modèles, ce qui nécessite des techniques de sélection ou de récupération (comme le RAG) dont l'efficacité dépend fortement des heuristiques employées. Ensuite, l'analyse des erreurs reste difficile : les trajectoires sont longues, complexes, et peu structurées, rendant l'identification des points de défaillance peu systématique. Par ailleurs, les agents actuels peinent à planifier efficacement leurs actions sur des tâches longues ou multi-étapes. Des erreurs de compréhension de l'environnement et des oublis d'instructions surviennent fréquemment, malgré les apports de méthodes comme ReAct ou Tree-of-Thought. La collecte de données reste également un frein important : les démonstrations humaines sont coûteuses, et les datasets existants manquent parfois de diversité ou d'annotations riches. Enfin, malgré les efforts de standardisation, la recherche reste fragmentée, avec des outils, formats et procédures d'évaluation hétérogènes, limitant la reproductibilité et la comparaison directe des approches. Ces constats motivent les contributions présentées dans la suite de ce mémoire, visant à mieux structurer les trajectoires, faciliter leur analyse, et explorer de nouvelles stratégies d'interaction plus robustes et explicables.

CHAPITRE 3 AGENTLAB : UNE SUITE D'OUTILS POUR L'ÉVALUATION DES AGENTS WEB

Ce chapitre présente AgentLab, le coeur de la contribution de ce mémoire. AgentLab constitue une suite d'outils complémentaires intégrée à l'écosystème BrowserGym pour faciliter le développement, l'évaluation, et l'analyse approfondie d'agents web pilotés par LLM. On présentera dans les sections qui suivent les quelques fonctionnalités du framework, qui visent à pallier les problèmes de reproducibilité évoqués précédemment.

3.1 Lancement d'une expérience

AgentLab simplifie considérablement le processus de configuration, de lancement d'expérimentations grâce à une interface simple et structurée. Cette section détaille les étapes essentielles permettant de configurer et d'exécuter efficacement une étude complète.

3.1.1 Gérer les expériences

La première étape pour lancer une expérimentation dans AgentLab consiste à définir clairement les paramètres nécessaires à l'expérience. Typiquement, cela inclut :

- Le benchmark à utiliser, par exemple MiniWoB++, WebArena, WorkArena.
- Les paramètres spécifiques des agents évalués (modèles de langage, prompts, méthodes d'interaction).

AgentLab encapsule ces modalités dans un interface simple et accessible. Plus généralement, afin de structurer et simplifier la démarche expérimentale, AgentLab propose un système intégré permettant à l'utilisateur de définir, gérer et suivre facilement les paramètres d'une expérience, de sa configuration à son exécution, jusqu'à la sauvegarde de ses données. Cette structuration favorise ainsi la rigueur méthodologique et la reproductibilité des expérimentations.

3.1.2 Exécution parallèle des expériences

Certains benchmarks regroupent un grand nombre de tâches relativement courtes, impliquant chacune une faible charge de calcul. Dans ce contexte, évaluer un agent nécessite d'exécuter ces nombreuses tâches, ce qui peut représenter une durée totale conséquente si elles sont traitées séquentiellement. AgentLab propose une solution en permettant la parallélisation

efficace des expériences. Celle-ci est rendu possible via des librairies telles que `ray` [43] ou `joblib` [44], qui permettent ainsi d'optimiser l'utilisation des ressources matérielles disponibles. Ce système permet ainsi une exécution exploitant au maximum les ressources matérielles disponibles, rendant les expérimentations rapides et efficaces, indépendamment de la complexité ou de l'échelle des benchmarks utilisés.

3.1.3 Graphes de dépendances

Certains benchmarks définissent des dépendances entre les tâches, imposant un ordre d'exécution spécifique. Par exemple, les tâches de WebArena doivent être réalisées dans une séquence prédéfinie pour garantir leur bon déroulement. De plus, certaines tâches appartenant à un même domaine (e.g., Reddit, Wikipedia, etc.) peuvent interférer entre elles. Par exemple, si deux tâches de commande sont exécutées simultanément sur un site d'e-commerce, elles risquent de partager le même panier d'achat, entraînant des résultats incohérents en raison d'une base de données commune.

Pour résoudre ce problème, AgentLab s'appuie sur `ray` afin d'implémenter des graphes de dépendances entre les tâches. Chaque benchmark peut ainsi être accompagné d'un graphe spécifiant l'ordre optimal d'exécution. Ce système garantit que chaque tâche attende l'achèvement de ses prédécesseurs avant de s'exécuter, tout en maintenant un niveau élevé de parallélisme lorsque cela est possible. Cette approche permet d'optimiser l'évaluation des agents en réduisant considérablement les temps d'exécution. Toujours dans le but de simplifier les évaluations, AgentLab s'occupe aussi de réinitialiser les benchmarks entre deux agents si nécessaire.

3.1.4 Gestion des temps d'attente (*timeouts*)

Certains benchmarks proposent des environnements particulièrement exigeants en termes de ressources computationnelles. Les fonctions de validation des tâches peuvent nécessiter une exploration approfondie des bases de données des sites évalués, tandis que les délais de réponse des sites web eux-mêmes peuvent ralentir considérablement les expérimentations. De même, les API de complétion de texte peuvent créer de nombreux ralentissements, dépendamment du fournisseur. Pour gérer ces contraintes, AgentLab utilise `ray` pour implémenter un mécanisme de garde-fou, permettant d'interrompre automatiquement une tâche dépassant une durée prédéfinie. Cette limite est paramétrée à la création d'une étude en fonction du temps alloué par action. Par exemple, une tâche avec un maximum de 10 interactions et une limite de 30 secondes par action sera arrêtée au bout de 300 secondes, garantissant ainsi une gestion efficace des ressources et évitant les blocages inutiles.

3.1.5 Gestion automatique des erreurs

Étant donné la nature stochastique des outils utilisés, certaines tâches peuvent rencontrer des erreurs lors de leur exécution. Parmi les erreurs les plus courantes figurent les défaillances d'API. Par exemple, le site d'OpenRouter [45] met à disposition des fournisseurs de complétion hébergeant des modèles de langage et vendant leurs inférences. Cependant, certains de ces fournisseurs peuvent être instables, entraînant des erreurs dans les réponses générées, ce qui perturbe le bon déroulement des expériences. De même, les dépassements de temps d'attente sont également considérés comme des erreurs impactant l'exécution des tâches.

Pour assurer la robustesse et la fiabilité des expérimentations, AgentLab propose un mécanisme automatisé de gestion des erreurs. Ce mécanisme permet d'identifier les épisodes incomplets ou échoués, et de les relancer. Une fois toutes les tâches exécutées, le système vérifie leur bon déroulement. En cas d'échec, les tâches concernées sont regroupées et réexécutées, garantissant ainsi une robustesse accrue des expériences, et la cohérence des résultats. La fréquence et le nombre de ces tentatives de récupération sont paramétrables selon les besoins de l'expérience.

Cette gestion automatisée des erreurs permet ainsi de maintenir une qualité élevée et constante des résultats, tout en réduisant significativement l'intervention manuelle nécessaire pour surveiller et corriger les problèmes rencontrés durant les expérimentations.

3.1.6 Sauvegarde et analyse des résultats

Après la réalisation des expériences, AgentLab sauvegarde les résultats de manière structurée, en incluant les détails de chaque interaction avec l'environnement. Ces traces peuvent être analysées ultérieurement à l'aide de l'outil AgentXRay, qui permet une inspection approfondie du comportement des agents, facilitant ainsi la compréhension des résultats et la validation des hypothèses expérimentales. Cette approche structurée garantit la reproductibilité, la transparence et l'efficacité des expériences menées avec AgentLab.

3.2 AgentXRay et analyse de traces

Pour permettre une analyse approfondie du comportement des agents, AgentLab intègre AgentXRay, un outil interactif basé sur la librairie Gradio [46] conçu pour explorer en détail les traces enregistrées au cours des expériences. Cette interface offre un accès à chacune des expériences présentent dans le dossier désigné. Une fois une expérience sélectionnée, AgentXRay donne une vue complète des tâches et germes aléatoires de l'étude. Il permet alors

pour chaque tâche d'observer en détail les interactions de l'agent avec son environnement, en fournissant un accès structuré aux informations essentielles telles que le profil d'exécution, l'objectif assigné à l'agent, les observations reçues à chaque étape, les actions entreprises et les raisonnements internes générés par le modèle. L'utilisateur peut ainsi suivre le processus décisionnel étape par étape et mieux comprendre les choix effectués par l'agent dans différentes situations. En observant précisément le moment où une erreur survient ou où une décision sous-optimale est prise, il devient possible d'identifier les limites des modèles utilisés et de les améliorer en conséquence. Cette granularité d'analyse facilite également la comparaison entre différentes approches et permet d'évaluer leur robustesse face aux défis posés par les environnements web complexes.

Grâce à tous ces outils, AgentXRay permet d'affiner le comportement d'un agent et d'optimiser les stratégies de navigation au sein des tâches. La figure 3.1 illustre l'interface d'AgentXRay.

3.3 Reproductibilité

La reproductibilité est un enjeu majeur dans l'évaluation des agents web, en raison des nombreuses sources de variabilité inhérentes aux environnements numériques. Les agents sont stochastiques par nature, et les tâches web ne sont pas strictement déterministes, ce qui signifie que des variations peuvent survenir même lors de l'exécution répétée d'une même expérience. AgentLab met en place plusieurs mécanismes pour rendre l'évaluation des agents aussi reproductibles et fiables que possible. Cette section explore les principaux défis de reproductibilité et les solutions mises en œuvre.

3.3.1 Unification du traitement web

L'un des obstacles à la reproductibilité des expériences sur les agents web est la différence d'implémentation dans le traitement des pages web entre différents moteurs et frameworks. Chaque moteur de recherche et bibliothèque de scraping (Playwright, Selenium, etc.) peut interpréter et structurer le DOM d'un site différemment, ce qui entraîne des variations dans les observations reçues par les agents et, potentiellement, dans leurs décisions. Par conséquent, l'évaluation de deux agents provenant de deux implémentations différentes s'en trouve biaisée.

AgentLab, en conjonction avec BrowserGym, assure une unification du traitement web en normalisant le format des observations et des actions. Chaque benchmark est livré avec un ensemble d'actions par défaut. Le format des observations est aussi fourni par défaut, cependant le pré-traitement des observations et les actions disponibles sont personnalisables

FIGURE 3.1 Explication visuelle de l'interface d'AgentXRay. AgentXRay est un outil d'analyse puissant dans l'analyse et le design d'agent dans AgentLab.

par l'utilisateur au besoin. Grâce à cette approche, les agents reçoivent des observations cohérentes, indépendamment de l'environnement d'exécution, garantissant ainsi des conditions expérimentales identiques.

3.3.2 Suivi des versions logicielles

Les évolutions logicielles constituent une autre source majeure de variabilité. Les mises à jour des bibliothèques utilisées pour l'exécution des agents (notamment Playwright, PyTorch, APIs des modèles) peuvent introduire des changements subtils affectant les résultats des expériences. De même, le système d'exploitation (Linux, Windows, etc.) et l'architecture (x64, AArch64, etc.) sont des facteurs qui peuvent impacter le déroulement d'une expérience. AgentLab et BrowserGym étant des librairies publiques vouées à évoluer dans le temps, il n'est pas envisageable de fixer les versions de toutes ces librairies et modalités.

Pour pallier cette situation, AgentLab suit et enregistre précisément les versions de chaque dépendance logicielle au moment du lancement d'une étude. Ceci permet non seulement d'identifier l'origine de variations dans les résultats, mais aussi de réexécuter des expériences dans un environnement identique en cas de besoin.

3.3.3 Gestion de la variabilité des modèles et des environnements

Les modèles de langage et leurs API évoluent fréquemment, avec des mises à jour pouvant modifier leur comportement. De plus, certains modèles propriétaires peuvent être sujet à des modifications, comme du *fine-tuning*. Ainsi la performance d'agents basés sur ces modèles peut varier sans raison apparente du point de vue de l'évaluateur. De même, les environnements web présentés dans les benchmarks peuvent être dynamiques, avec des modifications de l'interface utilisateur ou des données disponibles. La combinaison de ces facteurs accroît l'incertitude et complexifie l'évaluation des agents.

Pour mesurer ces dérives, AgentLab intègre un journal de reproductibilité, qui enregistre les expériences effectuées. Chaque entrée du journal conserve les données d'une expérience, comme le nom de l'agent, les versions de BrowserGym et AgentLab, le benchmark, les performances ainsi qu'un identifiant. Cela permet non seulement d'examiner l'évolution du comportement des modèles et des benchmarks au fil du temps, mais de pouvoir revenir en arrière et analyser les précédentes expériences.

3.3.4 Évaluation des variations avec `ReproducibilityAgent`

Au-delà du suivi des expériences, il est crucial de pouvoir contrôler l'impact des variations des environnements sur les performances des agents. En effet, le développement et les modifications fréquentes des librairies utilisées, mais aussi les variations dans les instances peuvent causer des changements dans le comportement des benchmark.

En plus de son agent par défaut `GenericAgent`, AgentLab propose `ReproducibilityAgent`, un agent dédié à l'évaluation des variations dans les benchmarks et instances web. Cet agent prend en entrée les traces d'une expérience faite avec l'agent de base `GenericAgent`, et roule exactement les mêmes actions sur les mêmes tâches. En comparant les résultats obtenus et les observations reçues, il est possible d'identifier les facteurs externes qui ont évolués dans le contenu d'un benchmark.

3.3.5 Suivi des performances avec le tableau de classement

Le tableau de classement [1] proposé par AgentLab joue un rôle clé pour suivre et comparer les performances des différents agents. Il centralise les résultats obtenus sur l'ensemble des benchmarks disponibles, offrant ainsi une vue synthétique des capacités des agents dans diverses conditions expérimentales. En regroupant les résultats de manière systématique, cet outil permet aux chercheurs de surveiller l'évolution des performances des agents au fil du temps, d'identifier les améliorations significatives ainsi que de détecter les éventuelles régressions causées par des variations externes. Le tableau de classement constitue une ressource essentielle pour évaluer rigoureusement et objectivement l'efficacité des stratégies implémentées au sein d'AgentLab.

Grâce à ces mécanismes, AgentLab offre une reproductibilité accrue des expériences, en limitant les variations dues aux différentes implémentations, aux modèles et aux environnements, tout en offrant des outils de suivi et d'analyse pour mieux comprendre l'impact de ces variations.

3.4 Création d'agents dans AgentLab

L'un des points clés d'AgentLab est de faciliter et de rendre flexible la conception des agents web, en s'appuyant sur `BrowserGym`. Un agent interagit avec un environnement web en recevant des observations et en produisant des actions. La structure d'un agent est définie en héritant de la classe `bgym.Agent` et en implémentant les méthodes essentielles qui permettent

1. `https://huggingface.co/spaces/ServiceNow/browsergym-leaderboard`

cette interaction.

3.4.1 Implémentation d'un agent

La création d'un agent au sein d'AgentLab repose sur une structure modulaire qui permet de définir, de manière claire et flexible, les éléments essentiels nécessaires à son fonctionnement. L'architecture d'un agent repose principalement sur trois composantes fondamentales, proposée par la classe `Agent` de BrowserGym :

- Définition d'un ensemble d'actions (`action_set`) : L'agent doit être capable de comprendre et d'interpréter un ensemble d'actions possibles dans l'environnement donné. Ces actions représentent les interactions que l'agent peut entreprendre pour accomplir ses objectifs.
- Pré-traitement des observations (`obs_preprocessor`) : L'agent reçoit des informations brutes provenant de l'environnement web. Afin de pouvoir les exploiter efficacement, ces observations doivent être transformées en un format adapté aux capacités de l'agent. Cela peut inclure la conversion d'images en texte, la structuration d'informations non organisées ou encore l'application de techniques spécifiques comme le marquage de caractéristiques visuelles.
- Prise de décision (`get_action`) : En fonction des observations traitées, l'agent doit déterminer une action appropriée à entreprendre. Ce processus de décision peut être basé sur des règles prédéfinies, des modèles d'apprentissage automatique, ou une combinaison de plusieurs stratégies.

BrowserGym fournit des outils standards pour le pré-traitement des observations et la définition d'ensembles d'actions. Toutefois, AgentLab permet également aux utilisateurs de personnaliser ces composants en fonction des besoins spécifiques de leurs expériences. Par exemple, un pré-traitement spécifique peut être appliqué pour extraire des caractéristiques pertinentes à partir d'images ou de textes fournis par l'environnement.

La capacité d'un agent à prendre des décisions repose sur sa capacité à combiner efficacement ces trois composantes. AgentLab permet de structurer ces interactions de manière cohérente, facilitant ainsi le développement d'agents robustes capables de s'adapter à une variété de scénarios expérimentaux. La section 3.7 montre deux implémentations d'agents via AgentLab.

3.4.2 Traçabilité et analyse du raisonnement de l'agent

Afin d'analyser efficacement les décisions prises par les agents et d'offrir une compréhension approfondie de leurs comportements, AgentLab fournit un système intégré permettant de

sauvegarder et structurer diverses informations utiles à l'analyse des expérimentations. Ce système permet notamment d'enregistrer :

- Le raisonnement interne de l'agent, qui explicite les étapes logiques suivies pour prendre ses décisions. Cela facilite l'identification des points forts et faibles de la stratégie adoptée par l'agent.
- Les interactions complètes avec les modèles de langage utilisés, incluant les requêtes et les réponses obtenues. Cela garantit une traçabilité précise et une analyse détaillée des échanges effectués avec ces modèles, en incluant les erreurs d'analyse syntaxique (*parsing*) et les corrections du modèle.
- Des métriques de performance détaillées, enregistrées tout au long des expérimentations. Ces métriques sont ensuite exploitées pour effectuer des analyses comparatives, permettant une évaluation rigoureuse et approfondie de l'efficacité des approches mises en œuvre. Ces métriques peuvent par exemple inclure les nombres de jetons ou le coût API des interactions de l'agent.

De plus, AgentLab offre la possibilité d'intégrer et d'afficher des informations personnalisées pertinentes pour chaque agent. Ces informations supplémentaires peuvent être présentées dans différents formats (texte enrichi, HTML), offrant une grande flexibilité pour visualiser et organiser les résultats expérimentaux de manière adaptée aux besoins spécifiques des utilisateurs. Celles-ci sont finalement accessibles dans AgentXRay pour observer en détail le comportement des agents.

Ce dispositif contribue significativement à la transparence et à la précision des analyses effectuées sur les agents évalués.

3.4.3 Configuration modulaire et flexible des agents

Afin de permettre une grande flexibilité dans la définition et l'expérimentation de différentes stratégies, AgentLab fournit un mécanisme modulaire de configuration des agents. Ce mécanisme permet aux utilisateurs de spécifier facilement des options clés influençant directement le comportement et les capacités de leurs agents. Cette approche permet notamment de configurer d'une part le type de raisonnement utilisé par l'agent, par exemple en activant ou non des stratégies avancées telles que le raisonnement *Chain-of-Thought*. D'autre part, elle configure la nature et le format des observations traitées, permettant de choisir entre plusieurs représentations (par exemple, HTML ou arbre d'accessibilité) en fonction du contexte ou du benchmark employé. Ce mécanisme offre également la possibilité d'adapter dynamiquement les paramètres des agents en fonction des caractéristiques spécifiques des benchmarks

évalués. Ainsi, l'agent peut ajuster automatiquement sa configuration pour répondre aux particularités de chaque environnement d'évaluation.

Toutes ces configurations, clairement définies au début d'une expérimentation, sont systématiquement enregistrées avec les résultats expérimentaux. Cela garantit une parfaite traçabilité des conditions expérimentales utilisées et facilite la reproduction et l'analyse détaillée des expériences menées. Grâce à cette approche modulaire, AgentLab encourage ainsi l'exploration rapide de différentes configurations et stratégies, tout en assurant rigueur méthodologique, reproductibilité et facilité d'analyse des résultats.

3.5 Évaluation et suivi des LLMs

AgentLab offre un mécanisme simple et flexible pour évaluer directement les LLMs sur des benchmarks spécifiques de BrowserGym, sans nécessiter une intégration complète au sein d'un agent. Cette approche permet de tester rapidement un LLM en le soumettant à des tâches variées, facilitant ainsi l'analyse de sa performance sur des questions de raisonnement sur le web. L'évaluation des LLMs est conçue pour être aussi fluide que possible : AgentLab fournit des interfaces standards permettant d'incorporer facilement un LLM existant et de mesurer ses performances sans avoir à modifier significativement la structure des agents. Cette simplicité permet aux utilisateurs de comparer rapidement plusieurs modèles, d'ajuster leurs configurations, et d'observer l'impact de ces modifications en temps réel.

En complément de cette évaluation simplifiée, AgentLab intègre un système de suivi des coûts des interactions avec les LLMs, sur les API d'OpenAI et d'OpenRouter. Ce suivi repose sur un gestionnaire de contexte, qui permet de mesurer précisément les ressources consommées par chaque requête. Cela fournit une visibilité claire sur les compromis entre performance et coût, facilitant l'optimisation des stratégies d'utilisation des modèles.

AgentLab permet ainsi une évaluation rapide et rigoureuse des LLMs, offrant aux utilisateurs un cadre cohérent pour évaluer et affiner leurs modèles.

3.6 Analyse d'erreurs automatique

Étant donné la quantité importante d'expériences réalisées dans AgentLab, l'analyse manuelle et exhaustive de chaque épisode peut s'avèrer particulièrement longue, voire impossible. Cette difficulté rend indispensable l'automatisation de l'analyse pour assurer un suivi rigoureux, cohérent et efficace des performances des agents.

Afin de pallier ce problème et d'améliorer la compréhension des comportements des agents

web développés, AgentLab introduit un pipeline dédié à l'analyse des erreurs commises par ces agents durant leurs interactions avec divers environnements web. Ce pipeline d'analyse automatique a pour objectif d'offrir une vision structurée et détaillée des points de défaillance et des erreurs commises par les agents.

Le pipeline s'articule principalement autour de deux composantes :

- `ChangeSummarizer` : Cette classe est conçue pour analyser les transitions entre chaque étape d'un épisode d'interaction de l'agent. Elle identifie et résume les différences notables dans les actions et les états de l'agent entre ces étapes consécutives. L'objectif principal est de détecter les anomalies ou les divergences dans le comportement de l'agent qui pourraient indiquer des erreurs ou des incohérences. En fournissant un résumé clair de ces transitions, le `ChangeSummarizer` aide à isoler les moments précis où des problèmes surviennent, facilitant ainsi une analyse plus ciblée et efficace des comportements de l'agent. Il permet par la même occasion de fournir des entrées de taille acceptable pour l'analyse global de l'épisode.
- `EpisodeSummarizer` : Cette classe prend en entrée les résumés générés pour chaque transition et les utilise pour effectuer une analyse à l'échelle de l'épisode complet. Elle se concentre sur la classification des catégories d'erreurs observées dans l'épisode, en identifiant les motifs récurrents ou les types d'erreurs spécifiques. L'`EpisodeSummarizer` offre ainsi une vue d'ensemble des performances de l'agent sur une série d'actions, permettant de mettre en évidence les domaines nécessitant des améliorations ou des ajustements. Cette approche globale est essentielle pour comprendre les tendances générales dans le comportement de l'agent et pour orienter les efforts de développement vers des solutions plus robustes et efficaces.

Les résultats fournis par ces deux classes sont stockées au format JSON avec les journaux (*logs*) de l'épisode. Ils sont alors disponibles dans l'un des onglets d'AgentXRay.

Cette API d'analyse des erreurs est un exemple des possibilités offertes par AgentLab en terme d'analyse d'agents. Son intégration offre la possibilité d'améliorer significativement les capacités d'AgentLab en matière de débogage, d'évaluation rigoureuse et de raffinement des agents web. De tels outils visent à offrir aux chercheurs et développeurs un moyen efficace et fiable – c'est-à-dire produisant des analyses cohérentes et reproductibles – pour diagnostiquer les limitations des modèles et orienter les futures améliorations méthodologiques et techniques.

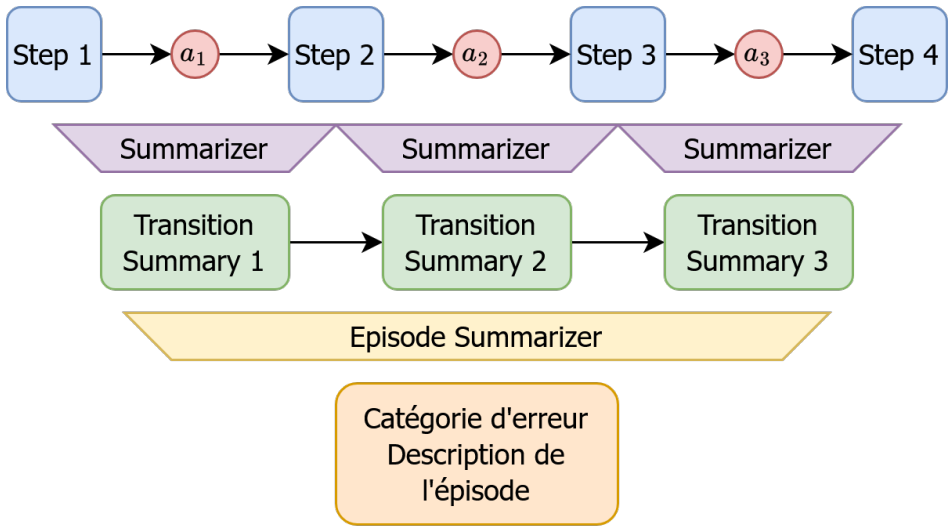


FIGURE 3.2 Schéma représentatif du fonctionnement du pipeline d'analyse d'erreur automatique.

3.7 Exemple de création de nouveaux agents

Dans cette section, nous présentons deux agents développés à l'aide d'AgentLab : WrapperAgent et WebDreamer. Ces deux agents témoignent de la flexibilité d'AgentLab, qui permet d'explorer rapidement des approches variées, qu'elles soient analytiques, heuristiques ou apprenantes, tout en restant compatibles avec l'écosystème BrowserGym. Chaque agent peut ainsi être facilement testé, visualisé et comparé aux approches existantes via une interface unifiée.

3.7.1 Wrapper agent

WrapperAgent est un agent wrapper construit avec AgentLab. Il agit comme un filtre autour d'un autre agent existant, qu'il appelle pour chaque observation. Il intercepte les observations entrantes, les prétraite au besoin, puis appelle l'agent sous-jacent pour produire une action. Cette action est ensuite post-traitée et renvoyée par l'agent. Chaque étape du modèle renvoie des informations qui sont agrégées (`agent_info`) et retournées avec chaque action. Ce design généraliste se veut agnostique de l'agent sous-jacent. Ainsi le wrapper se place par dessus n'importe quel agent AgentLab pour en modifier le comportement.

Ce type d'architecture ouvre la voie à de nombreuses possibilités :

- On peut, par exemple, intercepter les actions pour en extraire une intention, puis comparer cette intention au résultat de l'action pour vérifier et valider le résultat de l'action,

```python
class WrapperAgent(bgym.Agent):
    def __init__(self, agent_args: AgentArgs, chat_model_args: BaseModelArgs):
        # On initialise l'agent sous-jacent
        self.agent = agent_args.make_agent()
        self.chat_llm = chat_model_args.make_model()
        self.presave = []
        self.postsave = []

    def get_action(self, obs: Any) -> tuple[str, AgentInfo]:
        # L'agent analyse et modifie obs au besoin
        preprocessed_obs, preprocess_info = self.preprocess_agent(obs)
        self.presave.append(preprocess_info)

        # L'obs modifiée est passée à l'agent sous-jacent
        action, agent_info = self.agent.get_action(preprocessed_obs)

        # L'action proposée est analysée et modifiée
        post_processed_action, postprocess_info = self.postprocess_agent(action, obs)
        self.postsave.append(postprocess_info)

        agent_info = self.make_info(agent_info, preprocess_info, postprocess_info)

        return post_processed_action, agent_info
```

FIGURE 3.3 Implémentation minimaliste d'agent wrapper dans AgentLab

et modifier les instructions en conséquence à l'action suivante.

- Il est possible d'analyser l'action de sortie et le raisonnement du sous agent pour détecter un comportement dangereux, pour ainsi le soumettre à des règles et contraintes de sécurité.
- Le wrapper peut labelliser les données en temps réel lors de l'inférence, pour accélérer une éventuelle tâche de fine-tuning.
- Plus généralement, cette conception peut servir de base pour des agents plus généraux, comme des multi-agents hiérarchiques ou récursifs.

La figure 3.3 montre une version simplifiée du `WrapperAgent`, et en illustre la simplicité.

3.7.2 WebDreamer agent

AgentLab permet aussi l'implémentation d'agents existants. La figure 3.4 montre une implémentation minimaliste de l'agent WebDreamer [47]. WebDreamer est un agent dit *model-based*, c'est-à-dire qu'il s'appuie sur un modèle interne de l'environnement pour anticiper les conséquences de ses actions et améliorer son raisonnement. Il implémente plus précisément un algorithme de *Model Predictive Control* (MPC) [48], une approche classique en contrôle optimal. À chaque étape, l'agent génère un ensemble d'actions candidates, puis sélectionne un sous-ensemble prometteur. Pour chacune de ces actions, il simule une trajectoire future en s'appuyant sur son modèle interne, qui prédit les observations et résultats associés. Les

trajectoires sont ensuite évaluées à l'aide d'une fonction de score, et l'action initiale menant à la meilleure trajectoire est finalement sélectionnée et exécutée dans l'environnement réel. En suivant l'implémentation originale, le modèle prédit uniquement l'état suivant l'action, sous la forme d'un changement d'état. Le modèle prédit une description sémantique des changements qui auraient lieux dans l'environnement.

```python
class WebDreamerAgent(bgym.Agent):
    def get_action(self, obs: dict) -> tuple[str, AgentInfo]:
        self.history.append(obs)
        with set_tracker() as global_tracker:
            trackers = []

            # call Controller for possible actions
            with set_tracker("controller") as controller_tracker:
                possible_actions, content, markdown, stats = self.controller(self.history)
            trackers.append(controller_tracker)

            # call Refiner to select from possible actions
            with set_tracker("refiner") as refiner_tracker:
                refined_actions, content, markdown, stats = self.refiner(
                    possible_actions, self.history
                )
            trackers.append(refiner_tracker)

            # call WorldModel to predict state changes
            with set_tracker("world_model") as world_model_tracker:
                world_model_output, content, markdown, stats = self.world_model(
                    refined_actions, self.history
                )
            trackers.append(world_model_tracker)

            # call ValueModel to predict the value of the resulting states
            with set_tracker("value_model") as value_model_tracker:
                value_model_output, content, markdown, stats, txt_values = self.value_model(
                    world_model_output, refined_actions, self.history
                )
            trackers.append(value_model_tracker)

        # get all tracker stats
        stats_summary = global_tracker.stats
        for tracker in trackers:
            stats_summary.update(tracker.stats)

        action = self.get_best_action(value_model_output, refined_actions)
        agent_info = self.make_info(stats_summary)

        return action, agent_info
```

FIGURE 3.4 Implémentation minimaliste de l'agent WebDreamer dans AgentLab

L'implémentation de ces agents a été grandement facilitée par l'architecture modulaire d'AgentLab. La plateforme fournit une API unifiée pour les agents, les environnements et les modèles, ce qui permet d'intégrer facilement des stratégies avancées. Une fois l'agent implémenté, celui-ci devient directement compatible avec l'ensemble de l'écosystème : il peut être lancé sur n'importe quel benchmark BrowserGym, analysé avec AgentXRay, ou évalué dans des expériences parallélisées. Cette simplicité d'intégration permet de se concentrer sur les idées

de recherche plutôt que sur l'ingénierie logicielle, et accélère considérablement les cycles d'expérimentation.

3.8 Conclusion du chapitre

Ce chapitre a présenté en détail l'écosystème AgentLab, une suite complète d'outils intégrée à BrowserGym, spécialement conçue pour faciliter l'évaluation rigoureuse et reproductible d'agents web pilotés par des modèles de langage. À travers ses mécanismes de gestion des expériences, de parallélisation, de gestion des dépendances et d'analyse automatisée des erreurs, AgentLab répond aux défis méthodologiques rencontrés lors de l'évaluation d'agents dans des environnements web complexes. La flexibilité dans la conception et l'intégration d'agents, alliée à la transparence offerte par des outils comme AgentXRay, permet aux chercheurs et développeurs d'expérimenter aisément différentes stratégies et configurations. En simplifiant ainsi les aspects pratiques et techniques des expérimentations, AgentLab encourage une approche systématique et approfondie dans le développement d'agents web performants.

CHAPITRE 4 EXPÉRIMENTATIONS

Ce chapitre présente une évaluation expérimentale approfondie d'AgentLab sur plusieurs benchmarks de l'écosystème BrowserGym. L'objectif principal est de valider les fonctionnalités offertes par AgentLab, d'évaluer son efficacité à permettre des expérimentations rigoureuses et reproductibles, et d'explorer les capacités et les limites des agents web actuels. Les configurations expérimentales utilisées sont détaillées, suivies par une présentation des résultats quantitatifs obtenus avec divers modèles de langage. Une analyse des principales catégories d'erreurs observées est ensuite proposée, ainsi qu'une discussion sur des aspects pratiques tels que les coûts computationnels et les durées d'exécution. Cette évaluation permet ainsi de mettre en évidence les performances actuelles des agents web avancés ainsi que les axes d'amélioration prioritaires.

4.1 Configuration expérimentale et design de l'agent

Nous évaluons l'agent par défaut d'AgentLab, `GenericAgent`, sur l'ensemble des benchmarks disponibles dans l'écosystème BrowserGym. Cet repose sur un mécanisme de prompting modulaire, lui permettant d'exploiter divers outils pour optimiser ses réponses, notamment :

- Le raisonnement *Chain-of-Thought* (CoT)
- L'utilisation de captures d'écran avec *Set-of-Marks* (SoM)
- Un mécanisme de mémoire
- L'auto-critique et des exemples *in-context*

La configuration de l'agent détermine à la fois son comportement et le format d'observation, avec plusieurs options telles que l'utilisation des représentations HTML ou AXTree, la conservation de l'historique des observations et des actions passées, l'incorporation de captures d'écran ou d'images Set-of-Marks ou l'affichage d'informations HTML supplémentaires extraites de la page, comme la visibilité d'un élément ou encore ses coordonnées.

L'agent `ExampleAgent`, présenté en Figure 4.1, illustre une version simplifiée de `GenericAgent`. De plus, l'annexe A présente un exemple complet d'un prompt de l'agent.

En plus du prompting modulaire, `GenericAgent` met en œuvre une fonctionnalité de relance automatique en cas d'erreur de traitement par le LLM. Lorsqu'une erreur d'analyse syntaxique est détectée, l'agent relance une requête au LLM jusqu'à quatre fois. Si l'erreur persiste au-delà de ces quatre tentatives, la tâche est considérée comme échouée.

```python
from agentlab.agents import dynamic_prompting as dp

class ExampleAgent(Agent):
    def __init__(self, flags, chat_model_args):
        self.flags = flags
        self.action_set = HighLevelActionSet(["bid"])

        # API unifié pour l'usage des LLM/VLM
        # e.g.: OpenAI, OpenRouter, Azure ou TGI.
        self.chat_llm = chat_model_args.make_model()

    def get_action(self, obs):
        instructions = dp.GoalInstructions(obs["goal_object"])
        observation = dp.Observation(obs, self.flags.obs)
        action = dp.ActionPrompt(self.action_set,
        ↪  action_flags=self.flags.action)

        prompt = "\n".join([
            instructions.prompt, # le but de la tâche
            observation.prompt, # html et/ou ax_tree en fonction de
            ↪  flags.obs
            action.prompt, # une description de l'action space en
            ↪  fonction de flags.action
            action.concrete_ex, # un exemple du format de réponse désiré
        ])

        action_str = action.parse_answer(self.chat_llm(prompt))
        return action_str, bgym.AgentInfo(chat_messages=[prompt])

    def obs_preprocessor(self, obs):
        obs["processed_html"] = process_html(obs["dom_object"]
        return obs
```

```python
class ObsFlags:
    use_html: bool
    use_ax_tree: bool
    use_tabs: bool
    use_focused_element: bool
    use_error_logs: bool
    use_history: bool
    use_past_error_logs: bool
    use_action_history: bool
    use_screenshot: bool
    use_som: bool
    extract_visible_tag: bool
    extract_clickable_tag: bool
    extract_coords: bool
    filter_visible_elements_only:
    ↪  bool
    filter_with_bid_only: bool
    filter_som_only: bool

class ActionFlags(Flags):
    action_set:
    ↪  bgym.HighLevelActionSetArgs
    long_description: bool
    individual_examples: bool
```

FIGURE 4.1 Exemple d'implémentation d'un agent avec l'outil de prompting modulaire. Pour la simplicité de l'exemple, seulement une partie des fonctionnalités de `GenericAgent` sont montrées. La configuration de l'agent (`Flags`) est une `dataclass` conservée avec les traces de l'agent.

Les expériences utilisent la même configuration que celle de WorkArena++ [1], avec l'activation du paramètre `use_think_history`. Cette option permet à l'agent d'accéder à l'intégralité de son historique de raisonnement tout au long de l'exécution. Intuitivement la mémorisation des décisions passées permet des choix plus cohérents et informés dans la suite de l'épisode.

Nous avons évalué `GenericAgent` en utilisant un ensemble représentatif de modèles de langage état de l'art, incluant :

- GPT-4o et GPT-4o Mini [16] via l'API Azure OpenAI, avec respectivement les checkpoints 2024-08-06 et 2024-07-18.
- o1 Mini [49] via l'API OpenRouter, pour évaluer les performances d'un modèle basé sur le scaling du raisonnement (checkpoint 2024-09-12).
- Claude 3.5 Sonnet [12] via l'API OpenRouter, identifié comme le meilleur modèle sur l'ensemble des benchmarks (checkpoint 2024-10-22).
- Llama-3.1 70B et 405B [13] via l'API OpenRouter, représentants de la communauté

open-source, utilisés pour comparer leurs performances face aux modèles propriétaires et identifier des axes d'amélioration.

La principale métrique d'évaluation est le taux de succès des tâches, défini comme le pourcentage de tâches complétées avec succès par l'agent. Nous calculons également l'écart-type standard pour mesurer la fiabilité des résultats.

`GenericAgent` a été testé sur tous les benchmarks, avec quelques exceptions :

- WorkArena L3 : les résultats de WorkArena++ [1] ont été repris pour tous les modèles sauf Claude. Cette décision se base sur les résultats faibles atteint par Claude, le meilleur modèle, et par la volonté d'économiser des ressources budgétaires.
- VisualWebArena : seuls les modèles supportant les entrées multimodales ont été évalués, soit Claude, GPT-4o et GPT-4o Mini.
- Autres benchmarks : par souci d'équité, les entrées visuelles n'ont pas été utilisées lorsqu'elles n'étaient pas explicitement requises par le benchmark.

Les paramètres d'évaluation sont présentés dans le tableau 4.1. Chacun des benchmarks propose des germes aléatoires, curriculums ou groupements spécifiques. Ce paramétrage résulte en un nombre total de 5978 tâches, et un maximum de 99,390 interactions avec l'environnement.

Tableau 4.1 Récapitulatif des configurations expérimentales pour chaque benchmark.

Benchmark	**# Tâches**	**# Germes**	**Étapes max**	**# d'expériences**
MiniWoB	125	5	10	625
WebArena	812	/	30	812
VisualWebArena	910	/	30	910
WorkArena L1	33	10	30	330
WorkArena L2	341	curriculum	50	235
WorkArena L3	341	curriculum	50	235
WebLINX	31586	groupe de test	1	2650
Assistant Bench	214	/	30	181

4.2 Résultats

Le tableau 4.2 récapitule les expériences effectuées sur les benchmarks de BrowserGym et présente une vue d'ensemble permettant de comparer les capacités actuelles des LLM à gérer des interactions web complexes.

Le modèle Claude-3.5-Sonnet se distingue clairement avec des performances remarquablement élevées sur presque tous les benchmarks, notamment sur WorkArena L2 où il obtient une

performance exceptionnelle de 39.1%, comparativement à o1 Mini qui ne parvient qu'à 14.9%. Cette supériorité pourrait être expliquée par l'entraînement spécialisé de Claude 3.5 Sonnet en utilisation d'ordinateurs (*computer-use* [50]), ce qui pourrait lui conférer un avantage significatif dans le contexte web.

On observe également que les modèles Llama-3.1-70B et GPT-4o Mini affichent des performances relativement proches l'un de l'autre, suggérant que les modèles open-source tels que Llama restent dans une certaine mesure compétitifs face à leurs homologues commerciaux. Le modèle Llama-3.1-405B démontre également des résultats honorables, dépassant GPT-4o Mini sur la plupart des benchmarks. Ces résultats illustrent clairement le potentiel significatif des grands modèles open-source et la place importante qu'ils occuperont dans le domaine des agents web.

GPT-4o montre une amélioration notable de ses performances par rapport aux évaluations antérieures, probablement grâce à l'utilisation d'un nouveau checkpoint. Cette progression peut indiquer soit une progression des capacités de raisonnements de GPT-4o, soit la présence d'éléments de ces benchmarks dans la base de données utilisée pour améliorer le modèle.

Malgré ces progrès encourageants, les résultats sur AssistantBench restent globalement faibles pour tous les modèles évalués. Même le modèle Claude 3.5 Sonnet, pourtant très performant dans d'autres contextes, affiche une performance relativement modeste sur ce benchmark précis. Ceci s'explique probablement par le design spécifique d'AgentLab/BrowserGym, qui est adapté de façon généraliste aux tâches web. AssistantBench, axé spécifiquement sur les tâches de question/réponse, impose un cadre différent qui peut ne pas correspondre au contexte proposé par AgentLab.

TABLEAU 4.2 Tableau récapitulatif des expériences sur les benchmarks de BrowserGym. La colonne # Ep indique le nombre de tâches par benchmark. Pour des raisons de budget, seul Claude, le modèle le plus performant, a été lancé sur WorkArena L3. Les valeurs sont grisées et reflètent les résultats obtenu dans la publication originale de WorkArena++ [1]

Benchmark	# Ep.	Claude-3.5 -Sonnet	GPT-4o	GPT-4o Mini	Llama-3.1 70B	Llama-3.1 405B	o1 Mini
MiniWoB	625	**69.8** $\pm$1.8	63.8 $\pm$1.9	56.6 $\pm$2.0	57.6 $\pm$2.0	64.6 $\pm$1.9	67.8 $\pm$1.9
WorkArena L1	330	56.4 $\pm$2.7	45.5 $\pm$2.7	27.0 $\pm$2.4	27.9 $\pm$2.5	43.3 $\pm$2.7	**56.7** $\pm$2.7
WorkArena L2	235	**39.1** $\pm$3.2	8.5 $\pm$1.8	1.3 $\pm$0.7	2.1 $\pm$0.9	7.2 $\pm$1.7	14.9 $\pm$2.3
WorkArena L3	235	**0.4** $\pm$0.4	0.0 $\pm$0.0	0.0 $\pm$0.0	0.0 $\pm$0.0	0.0 $\pm$0.0	0.0 $\pm$0.0
WebLINX	2650	**13.7** $\pm$0.6	12.5 $\pm$0.6	11.6 $\pm$0.6	8.9 $\pm$0.5	7.9 $\pm$0.5	12.5 $\pm$0.6
WebArena	812	**36.2** $\pm$1.7	31.4 $\pm$1.6	17.4 $\pm$1.3	18.4 $\pm$1.4	24.0 $\pm$1.5	28.6 $\pm$1.6
VisualWebArena	910	21.0 $\pm$1.3	**26.7** $\pm$1.5	16.9 $\pm$1.2	-	-	-
AssistantBench	181	5.2 $\pm$1.5	4.8 $\pm$2.4	2.1 $\pm$1.0	2.8 $\pm$1.1	3.9 $\pm$1.0	**6.9** $\pm$2.2

4.3 Analyse des erreurs

Lors de l'évaluation des agents d'intelligence artificielle, les erreurs rencontrées peuvent généralement être classées en plusieurs catégories distinctes :

- **Erreurs de navigation** : Ces erreurs surviennent lorsque l'agent rencontre des difficultés à accéder à la bonne page, souvent dues à une mauvaise navigation sur la page web.
- **Erreurs de gestion des formulaires** : Elles concernent l'entrée incorrecte de données dans les formulaires ou l'incapacité à repérer les erreurs de soumission de ces formulaires.
- **Erreurs de compréhension de la tâche** : Ce type d'erreur apparaît lorsque l'agent interprète mal des consignes ambiguës ou imprécises, ce qui mène à des actions inadaptées ou incorrectes.
- **Comportement de blocage** : Ces erreurs se manifestent par la répétition persistante d'une même action infructueuse, sans ajustement ni changement de stratégie.
- **Échecs d'extraction d'informations** : Elles se produisent lorsque l'agent atteint le bon endroit mais ne réussit pas à extraire ou à exploiter correctement les informations nécessaires.
- **Erreurs externes** : Ces erreurs sont liées à des problèmes techniques indépendants du fonctionnement de l'agent, comme des pannes d'API, des interruptions réseau ou des dysfonctionnements système, affectant ainsi négativement sa performance.

Bien que ces catégories ne soient pas exhaustives, elles offrent un cadre pratique permettant d'identifier les faiblesses des agents et d'orienter les améliorations vers une plus grande fiabilité. Ce sont ces catégories qui sont utilisées comme classes par l'outil d'analyse d'erreurs automatisé mentionné en section 3.6.

Compte tenu de l'ampleur des expérimentations, cette étude propose une vue d'ensemble qualitative des erreurs rencontrées par les agents. Les traces détaillées des expériences sont néanmoins accessibles publiquement et permettent une exploration approfondie via l'interface AgentXRay. Bien que le pipeline d'analyse d'erreur ait été conçu et testé avec succès sur un sous-ensemble limité de tâches, son application exhaustive à l'ensemble des expérimentations reste à finaliser.

Les recherches futures devront ainsi porter sur la finalisation et l'application complète de ce pipeline automatisé, afin de produire une évaluation systématique et exhaustive, susceptible de fournir des résultats plus complets et fiables.

4.4 Analyse des coûts et durées

Cette section présente en détail les coûts associés aux expériences réalisées, ainsi que les durées d'exécution pour le modèle le plus performant testé, Claude 3.5 Sonnet, sur l'ensemble des benchmarks considérés.

4.4.1 Coûts

Le tableau 4.3 détaille les coûts totaux en dollars américains (USD), ainsi que le nombre de jetons utilisés en entrée et en sortie des différents modèles LLM évalués. Ce tableau montre certaines limitations des Agents Web :

- Le modèle le plus coûteux est o1 Mini, avec un total de 971.12$. À cause de sa méthode de raisonnement basé sur la mise à l'échelle de l'inférence, o1 Mini infère au total 38.11 millions de jetons de sortie, montrant l'une des limitations de ce type de modèle.
- Les modèles les plus performants sont les plus chers. De tous les LLMs testés, les modèles plus légers comme Llama-3.1 70B et GPT-4o Mini sont les moins performant.
- On aperçoit aussi une limite des modèles libres de droit comparés à leurs concurrents privés. Bien que proposés pour des prix équivalents par les fournisseurs de services, ces coûts ne se reflètent pas dans les performances des modèles.

Le tableau 4.3 met aussi en avant l'efficacité des modèles les plus performants. En effet, Claude, GPT-4o et o1 Mini ont des quantités de jetons d'entrée relativement basses, impliquant qu'ils ont tendances à aller directement vers une solution, en se perdant moins souvent dans l'environnement.

Agent	**Coût Total ($)**	**Jetons d'entrée (M de jetons)**	**Jetons de sortie (M de jetons)**	**Coût API d'entrée ($/M jetons)**	**Coût API de sortie ($/M jetons)**
Claude	894.75	276.20	4.41	3.00	15.00
GPT-4o	720.72	277.25	2.76	2.50	10.00
GPT-4o-mini	75.73	479.72	6.29	0.15	0.60
Llama-3.1 405B	846.45	359.89	8.13	2.30	2.30
Llama-3.1 70B	79.19	221.23	5.02	0.35	0.35
o1 Mini	971.07	171.25	38.11	3.00	12.00

Tableau 4.3 Résumé des coûts des modèles principaux utilisés dans les expériences. Les valeurs sont additionnées au travers de tous les benchmarks, à l'exception de VisualWebArena.

4.4.2 Durées d'exécution

Les durées d'exécution présentées ici correspondent au modèle le plus performant des expériences, Claude 3.5 Sonnet. Le tableau 4.4 résume les durées cumulées totales, ainsi que les durées moyennes par étape, pour les principaux benchmarks évalués. Ce tableau présente les métriques suivantes :

- **Durées cumulées (heures)** : Temps total nécessaire pour compléter les expériences sur un benchmark donné. Cette valeur ne tient pas compte de la parallélisation.
- **Durée moyenne (secondes)** : Temps moyen par interaction (étape) de l'agent au sein d'un benchmark.
- **Durée d'exécution cumulées (heures)** : Temps total d'exécution de l'environnement uniquement, excluant le temps de traitement de l'agent (par exemple, le temps de réponse des modèles LLM ou des API).
- **Durée d'exécution moyenne (secondes)** : Temps moyen passé dans l'environnement par étape, permettant d'évaluer l'impact des latences externes (API, chargement de pages web).
- **Nombre d'interactions** : Nombre total d'étapes effectuées par l'agent sur chaque benchmark. Une interaction correspond à une décision prise par l'agent, suivie de son exécution et de l'observation du nouvel état de l'environnement.

Benchmark	Durées Cumulées (heures)	Durée Moyenne (secondes)	Durées d'Exécution Cumulées (heures)	Durée d'Exécution Moyenne (secondes)	Nombre d'Interactions
AssistantBench	4.0	9.1	2.0	4.5	1590
MiniWoB	3.1	5.0	0.7	1.1	2282
WebArena	18.6	12.2	11.6	7.6	5493
WebLINX	4.5	6.1	0.1	0.2	2649
WorkArena L1	8.2	9.9	4.4	5.3	2985
WorkArena L2	23.6	10.7	10.9	4.9	7947
WorkArena L3	16.1	9.8	7.4	4.5	5884
VisualWebArena	16.9	14.1	10.5	8.8	4319

Tableau 4.4 Résumé des durées d'exécution cumulées et moyennes par étape, réalisées avec Claude 3.5 Sonnet, au travers de benchmarks. Les durées cumulées indiquent la durée totale des expériences sans prendre en compte la parallélisation. Les durées d'exécution désignent la durée d'exécution de l'environnement, par contraste avec celle des agents.

La plupart des expériences ont été réalisées sur des serveurs de calculs dotés de CPU Intel(R) Xeon(R) Gold 6126 2.60GHz, avec une RAM effective illimitée, mais elles pourraient être exécutées localement sur des ordinateurs portables standards, les contraintes principales

provenant généralement des environnements eux-mêmes (chargement de pages web, appels d'API distants, etc.).

Ces informations montrent clairement l'échelle et les ressources nécessaires pour mener des expériences à grande échelle sur les agents web, et servent de référence utile pour planifier de futures expériences similaires ou étendre les recherches à venir.

CHAPITRE 5 CONCLUSION

5.1 Synthèse des travaux

Ce mémoire s'inscrit dans le contexte en plein essor des agents web pilotés par des LLMs. Face à la fragmentation des benchmarks existants et à l'absence d'outils d'évaluation unifiés, nous avons introduit AgentLab, une suite méthodologique et technique conçue pour enrichir l'écosystème BrowserGym en répondant spécifiquement aux besoins croissants de reproductibilité, de scalabilité et d'analyse comportementale fine des agents web.

Les contributions principales sont les suivantes :

- La conception et la formalisation d'une interface unifiée et modulaire, permettant non seulement le lancement et la gestion, mais surtout l'analyse systématique et reproductible d'expérimentations à grande échelle. Cette contribution méthodologique facilite la standardisation des protocoles expérimentaux et favorise ainsi la reproductibilité des résultats dans la communauté.
- Le développement et l'intégration de mécanismes avancés de suivi et d'analyse comportementale (journeaux structurés, AgentXRay pour visualiser les décisions internes, journal détaillé de reproductibilité, gestion proactive des erreurs). Cette avancée permet une exploration interactive et approfondie des comportements des agents, répondant ainsi à un besoin critique d'interprétabilité et d'analyse fine dans le domaine.
- Une validation expérimentale rigoureuse effectuée sur plusieurs benchmarks représentatifs de BrowserGym, mettant en lumière des résultats concrets sur les capacités et limites actuelles des LLMs dans des tâches d'automatisation web complexes. Les expérimentations révèlent notamment des points de blocage persistants, ouvrant ainsi des perspectives précises pour les recherches futures dans l'amélioration des agents web.

En somme, le travail présenté contribue à structurer et enrichir l'évaluation expérimentale des agents web pilotés par des LLMs, tout en identifiant clairement les défis ouverts et les opportunités scientifiques qui en découlent. Ces travaux sont synthétisés par deux publications acceptées : d'une part, la publication scientifique sur l'écosystème BrowserGym dans la revue *Transactions on Machine Learning Research* (TMLR)[4], et d'autre part, la présentation de WorkArena++ à la conférence *Neural Information Processing Systems* (NeurIPS)[1].

5.2 Limitations de la solution proposée

Malgré les fonctionnalités avancées d'AgentLab et de l'écosystème BrowserGym, plusieurs limitations importantes subsistent. Certaines relèvent de choix techniques actuels, d'autres sont liées à la nature même des benchmarks sur lesquels les agents évoluent.

Reproductibilité : Bien que l'architecture d'AgentLab cherche à maximiser la reproductibilité des expériences (journal de reproductibilité, germes aléatoires, relance d'épisodes, etc.), de nombreux facteurs restent hors de contrôle. Les modèles commerciaux évoluent silencieusement, les sites web dynamiques changent de contenu ou de structure, et les systèmes backend peuvent réagir différemment selon le moment de l'exécution ou la localisation géographique.

Navigation libre sur le web : Certains benchmarks comme AssistantBench reposent sur la navigation libre sur le web. Laisser un agent interagir sans contrôle avec Internet pose de nombreuses questions en termes de sécurité, d'éthique, et de respect des conditions d'utilisation des sites. Un agent mal configuré pourrait, par exemple, interagir de manière abusive avec un service ou transmettre des informations sensibles.

Détection de bots : De nombreux sites sont protégés par des systèmes de détection automatique des robots (CAPTCHAs, restrictions IP, etc.). Ces protections rendent difficile, voire impossible, l'évaluation d'agents sur certaines tâches. AssistantBench est notamment très impacté par ce problème, nécessitant un accès au web pour répondre à ses problèmes. La capacité d'un agent à réussir une tâche peut dépendre autant de sa performance que de sa capacité à contourner ces barrières techniques.

Interférence entre agents : Certains environnements comme WebArena ou VisualWebArena reposent sur une instance unique de serveur backend, partagée par les différents agents. Il est donc impossible de faire évoluer plusieurs agents en parallèle sans risquer des conflits ou une corruption de l'état de la base de données. Cela ralentit considérablement les campagnes d'expérimentation et limite la scalabilité du système.

Exécution synchronisée dans BrowserGym : L'infrastructure actuelle de BrowserGym repose sur une boucle synchrone pour l'exécution des actions et des observations. Cela limite fortement la capacité à paralléliser les interactions, notamment dans les tâches complexes où les agents enchaînent de nombreuses étapes. Il en résulte des ralentissements importants et un sous-emploi des ressources disponibles, en particulier en environnement serveur multi-coeurs.

5.3 Améliorations futures

Plusieurs pistes d'évolution s'offrent pour faire progresser AgentLab vers une plateforme plus généraliste, efficace et robuste. Parmi elles, l'extension d'AgentLab au-delà du cadre strict de BrowserGym ouvre la possibilité d'en faire une interface unifiée adaptée à une grande variété d'environnements. Cette extension multi-domaines pourrait inclure des interfaces desktop, à l'instar d'OSWorld [30], ou encore des APIs REST, des outils (terminal Python). Des possibilités plus larges pourraient même inclure des simulateurs physiques ou 3D, avec des extensions en robotique. Pour cela, une abstraction plus flexible du cycle observation-action serait nécessaire afin de garantir la généralisation des outils d'évaluation et de débogage.

Par ailleurs, les traces collectées par AgentLab constituent déjà élément essentiel pour le déploiement de méthodes de *fine-tuning*. En capitalisant sur les observations, actions, raisonnements internes et résultats collectés lors des expériences, AgentLab rend possible l'entraînement de nouveaux modèles ou l'adaptation de modèles existants via apprentissage supervisé, distillation ou imitation learning. Ces données pourraient également nourrir des approches plus dynamiques comme l'apprentissage en ligne, permettant ainsi aux agents de s'améliorer de manière incrémentale au fil de leurs interactions. Une telle exploitation des traces favoriserait la création de boucles d'entraînement plus rapides, plus ciblées et plus efficaces que les approches traditionnelles.

Dans cette optique, l'automatisation de l'analyse d'erreurs représente un autre levier intéressant. Le développement d'outils capables de classifier les échecs en fonction de leur nature (raisonnement, navigation, extraction d'information, erreurs de timing, etc.) et de produire des diagnostics exploitables permettrait d'améliorer la compréhension des points de blocage rencontrés par les agents, et d'améliorer en retour les agents.

L'optimisation du temps d'exécution constitue également un axe important pour l'avenir d'AgentLab. Aujourd'hui, certains benchmarks restent coûteux à exécuter en raison des limites de parallélisation ou du fonctionnement synchrone de BrowserGym. Une exécution asynchrone pourrait réduire significativement les coûts et les durées des campagnes d'expérimentation.

Un autre enjeu majeur pour l'avenir des agents est la question de la sécurité. À mesure que ces systèmes gagnent en autonomie et en accessibilité, le risque de comportements malveillants ou détournés augmente. Un agent mal conçu pourrait, par exemple, exécuter des actions non désirées sur le web, envoyer des données sensibles ou interagir de manière abusive avec des services. De plus, les tentatives de *jailbreak*, visant à contourner les instructions ou les garde-fous imposés à un agent, représentent une menace réelle et déjà observée dans le

cas des modèles de langage. Il devient donc crucial d'intégrer des mécanismes de contrôle, de validation d'actions, et de surveillance des comportements, et de créer les benchmarks associés, afin de garantir que les agents restent dans un cadre d'exécution sûr et aligné avec les intentions de leurs concepteurs.

Enfin, AgentLab est premièrement un outil de développement d'agents web. La conception de nouveaux agents, avec de nouveaux mécanismes de réflexion est un axe central dans l'avenir des agents intelligents. L'intégration de mémoires persistantes pourrait permettre d'explorer des agents capables de capitaliser sur leurs interactions passées. En stockant certaines informations dans des journeaux internes ou des bases vectorielles, les agents pourraient améliorer leur cohérence sur plusieurs épisodes ou adapter leur comportement selon l'historique. Cette idée ouvre la possibilité de benchmarks continus et d'évaluations évolutives, dans lesquels l'apprentissage ne serait plus figé mais s'inscrirait dans une dynamique incrémentale, notion déjà considérée dans certains benchmarks comme WorkArena++.

RÉFÉRENCES

[1] L. Boisvert, M. Thakkar, M. Gasse, M. Caccia, T. de Chezelles, Q. Cappart, N. Chapados, A. Lacoste et A. Drouin, "Workarena++ : Towards compositional planning and reasoning-based common knowledge work tasks," *Advances in Neural Information Processing Systems*, vol. 37, p. 5996–6051, 2024.

[2] Q. Lhoest, A. V. del Moral, Y. Jernite, A. Thakur, P. von Platen, S. Patil, J. Chaumond, M. Drame, J. Plu, L. Tunstall, J. Davison, M. Šaško, G. Chhablani, B. Malik, S. Brandeis, T. L. Scao, V. Sanh, C. Xu, N. Patry, A. McMillan-Major, P. Schmid, S. Gugger, C. Delangue, T. Matussière, L. Debut, S. Bekman, P. Cistac, T. Goehringer, V. Mustar, F. Lagunas, A. M. Rush et T. Wolf, "Datasets : A community library for natural language processing," 2021. [En ligne]. Disponible : https://arxiv.org/abs/2109.02846

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang et W. Zaremba, "OpenAI Gym," 2016. [En ligne]. Disponible : https://arxiv.org/abs/1606.01540

[4] T. L. S. D. Chezelles, M. Gasse, A. Drouin, M. Caccia, L. Boisvert, M. Thakkar, T. Marty, R. Assouel, S. O. Shayegan, L. K. Jang, X. H. Lù, O. Yoran, D. Kong, F. F. Xu, S. Reddy, Q. Cappart, G. Neubig, R. Salakhutdinov, N. Chapados et A. Lacoste, "The browsergym ecosystem for web agent research," 2025. [En ligne]. Disponible : https://arxiv.org/abs/2412.05467

[5] T. Hastie, R. Tibshirani et J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY, USA : Springer New York Inc., 2001.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser et I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[7] R. S. Sutton et A. G. Barto, *Reinforcement Learning : An Introduction*, 2e éd. The MIT Press, 2018. [En ligne]. Disponible : http://incompleteideas.net/book/the-book-2nd.html

[8] Y. Lecun, L. Bottou, Y. Bengio et P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, p. 2278–2324, 1998.

[9] S. Hochreiter et J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, nov. 1997. [En ligne]. Disponible : https://doi.org/10.1162/neco.1997.9.8.1735

[10] I. Goodfellow, Y. Bengio et A. Courville, *Deep Learning*. MIT Press, 2016, book in preparation for MIT Press. [En ligne]. Disponible : http://www.deeplearningbook.org

[11] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever et D. Amodei, "Language models are few-shot learners," dans *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA : Curran Associates Inc., 2020.

[12] Anthropic, "Claude 3.5 Sonnet model card addendum," 2024. [En ligne]. Disponible : https://www-cdn.anthropic.com/fed9cc193a14b84131812372d8d5857f8f304c52/Model_Card_Claude_3_Addendum.pdf

[13] Meta, "The Llama 3 herd of models," 2024. [En ligne]. Disponible : https://arxiv.org/abs/2407.21783

[14] A. Radford et K. Narasimhan, "Improving language understanding by generative pre-training," 2018. [En ligne]. Disponible : https://api.semanticscholar.org/CorpusID:49313245

[15] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu et D. Amodei, "Scaling laws for neural language models," *CoRR*, vol. abs/2001.08361, 2020. [En ligne]. Disponible : https://arxiv.org/abs/2001.08361

[16] OpenAI, "GPT-4 technical report," 2024. [En ligne]. Disponible : https://arxiv.org/abs/2303.08774

[17] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, O. Vinyals, J. W. Rae et L. Sifre, "Training compute-optimal large language models," dans *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA : Curran Associates Inc., 2022.

[18] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, p. 24 824–24 837, 2022.

[19] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao et K. Narasimhan, "Tree of thoughts : Deliberate problem solving with large language models," *Advances in neural information processing systems*, vol. 36, p. 11 809–11 822, 2023.

[20] S. Yao, J. Zhao, D. Yu, I. Shafran, K. R. Narasimhan et Y. Cao, "ReAct : Synergizing reasoning and acting in language models," dans *NeurIPS 2022*

Foundation Models for Decision Making Workshop, 2022. [En ligne]. Disponible : https://openreview.net/forum?id=tvI4u1ylcqs

[21] G. Kim, P. Baldi et S. McAleer, "Language models can solve computer tasks," *Advances in Neural Information Processing Systems*, vol. 36, p. 39 648–39 677, 2023.

[22] O. Yoran, S. J. Amouyal, C. Malaviya, B. Bogin, O. Press et J. Berant, "AssistantBench : Can web agents solve realistic and time-consuming tasks ?" dans *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Y. Al-Onaizan, M. Bansal et Y.-N. Chen, édit. Miami, Florida, USA : Association for Computational Linguistics, nov. 2024, p. 8938–8968. [En ligne]. Disponible : https://aclanthology.org/2024.emnlp-main.505/

[23] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, p. 9459–9474, 2020.

[24] K. Guu, K. Lee, Z. Tung, P. Pasupat et M. Chang, "Retrieval augmented language model pre-training," dans *International conference on machine learning*. PMLR, 2020, p. 3929–3938.

[25] R. Nakano, J. Hilton, S. Balaji, J. Wu, L. Ouyang, C. Kim, C. Hesse, S. Jain, V. Kosaraju, W. Saunders, X. Jiang, K. Cobbe, T. Eloundou, G. Krueger, K. Button, M. Knight, B. Chess et J. Schulman, "WebGPT : Browser-assisted question-answering with human feedback," 2022. [En ligne]. Disponible : https://arxiv.org/abs/2112.09332

[26] X. Deng, Y. Gu, B. Zheng, S. Chen, S. Stevens, B. Wang, H. Sun et Y. Su, "Mind2web : Towards a generalist agent for the web," *Advances in Neural Information Processing Systems*, vol. 36, p. 28 091–28 114, 2023.

[27] H. He, W. Yao, K. Ma, W. Yu, Y. Dai, H. Zhang, Z. Lan et D. Yu, "WebVoyager : Building an end-to-end web agent with large multimodal models," dans *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1 : Long Papers)*, L.-W. Ku, A. Martins et V. Srikumar, édit. Bangkok, Thailand : Association for Computational Linguistics, août 2024, p. 6864–6890. [En ligne]. Disponible : https://aclanthology.org/2024.acl-long.371/

[28] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda et T. Scialom, "Toolformer : Language models can teach themselves to use tools," *Advances in Neural Information Processing Systems*, vol. 36, p. 68 539–68 551, 2023.

[29] Y. Du, F. Wei et H. Zhang, "Anytool : self-reflective, hierarchical agents for large-scale api calls," dans *Proceedings of the 41st International Conference on Machine Learning*,

ser. ICML'24. JMLR.org, 2024.

[30] T. Xie, D. Zhang, J. Chen, X. Li, S. Zhao, R. Cao, J. H. Toh, Z. Cheng, D. Shin, F. Lei *et al.*, "Osworld : Benchmarking multimodal agents for open-ended tasks in real computer environments," *Advances in Neural Information Processing Systems*, vol. 37, p. 52 040–52 094, 2024.

[31] D. Amodei, C. Olah, J. Steinhardt, P. F. Christiano, J. Schulman et D. Mané, "Concrete problems in AI safety," *CoRR*, vol. abs/1606.06565, 2016. [En ligne]. Disponible : http://arxiv.org/abs/1606.06565

[32] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg et D. Amodei, "Deep reinforcement learning from human preferences," *Advances in neural information processing systems*, vol. 30, 2017.

[33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford et O. Klimov, "Proximal policy optimization algorithms," *ArXiv*, vol. abs/1707.06347, 2017. [En ligne]. Disponible : https://api.semanticscholar.org/CorpusID:28695052

[34] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon et C. Finn, "Direct preference optimization : Your language model is secretly a reward model," *Advances in Neural Information Processing Systems*, vol. 36, p. 53 728–53 741, 2023.

[35] DeepSeek-AI, "Deepseek-r1 : Incentivizing reasoning capability in llms via reinforcement learning," 2025. [En ligne]. Disponible : https://arxiv.org/abs/2501.12948

[36] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. K. Li, Y. Wu et D. Guo, "Deepseekmath : Pushing the limits of mathematical reasoning in open language models," 2024. [En ligne]. Disponible : https://arxiv.org/abs/2402.03300

[37] E. Z. Liu, K. Guu, P. Pasupat, T. Shi et P. Liang, "Reinforcement Learning on Web Interfaces using Workflow-guided Exploration," dans *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018. [En ligne]. Disponible : https://openreview.net/forum?id=ryTp3f-0-

[38] S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, T. Ou, Y. Bisk, D. Fried, U. Alon et G. Neubig, "WebArena : A realistic web environment for building autonomous agents," 2024. [En ligne]. Disponible : https://arxiv.org/abs/2307.13854

[39] A. Drouin, M. Gasse, M. Caccia, I. H. Laradji, M. D. Verme, T. Marty, D. Vazquez, N. Chapados et A. Lacoste, "Workarena : how capable are web agents at solving common knowledge work tasks ?" dans *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.

[40] X. H. Lù, Z. Kasner et S. Reddy, "Weblinx : real-world website navigation with multi-turn dialogue," dans *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.

[41] A. D. Tur, N. Meade, X. H. Lù, A. Zambrano, A. Patel, E. Durmus, S. Gella, K. Stańczak et S. Reddy, "Safearena : Evaluating the safety of autonomous web agents," 2025. [En ligne]. Disponible : https://arxiv.org/abs/2503.04957

[42] M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. D. Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, H. Tan et O. G. Younis, "Gymnasium : A standard interface for reinforcement learning environments," 2024. [En ligne]. Disponible : https://arxiv.org/abs/2407.17032

[43] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan et I. Stoica, "Ray : A distributed framework for emerging AI applications," 2018. [En ligne]. Disponible : https://arxiv.org/abs/1712.05889

[44] Joblib Development Team, "Joblib : running python functions as pipeline jobs," 2020. [En ligne]. Disponible : https://joblib.readthedocs.io/

[45] OpenRouter, "Openrouter," 2025, accessed : 2025-03-30. [En ligne]. Disponible : https://openrouter.ai/

[46] A. Abid, A. Abdalla, A. Abid, D. Khan, A. Alfozan et J. Zou, "Gradio : Hassle-free sharing and testing of ml models in the wild," 2019. [En ligne]. Disponible : https://arxiv.org/abs/1906.02569

[47] Y. Gu, B. Zheng, B. Gou, K. Zhang, C. Chang, S. Srivastava, Y. Xie, P. Qi, H. Sun et Y. Su, "Is your llm secretly a world model of the internet ? model-based planning for web agents," *CoRR*, vol. abs/2411.06559, 2024. [En ligne]. Disponible : https://arxiv.org/abs/2411.06559

[48] C. E. García, D. M. Prett et M. Morari, "Model predictive control : Theory and practice—a survey," *Automatica*, vol. 25, nº. 3, p. 335–348, 1989. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/0005109889900022

[49] OpenAI, "OpenAI o1-mini : Advancing cost-efficient reasoning," https://openai.com/index/openai-o1-mini-advancing-cost-efficient-reasoning, 2024, [Online ; accessed 12-September-2024].

[50] Anthropic, "Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku," https://www.anthropic.com/news/3-5-models-and-computer-use, 2024, [Online ; accessed 22-October-2024].

ANNEXE A PROMPT DE GENERICAGENT

Voici un exemple complet de prompt produit par `GenericAgent` sur le benchmark WorkArena L2 :

Exemple de prompt de `GenericAgent`

```
# Instructions
Review the current state of the page and all other information to find the best
possible next action to accomplish your goal. Your answer will be interpreted
and executed by a program, make sure to follow the formatting instructions.

## Goal:

Managing Your Existing Expenses

 Concretely, you need to complete the following steps:
1. Navigate to the "Expense Lines" module of the "Cost" application.
2. Create a filter for the list to extract all entries where:
    - "Short description" contains "#SERIES-d1e0dc99-5"
3. Delete expense lines with duplicated short descriptions, keeping only the most expensive duplicate.

# Observation of current step:

## Currently open tabs:
Tab 0 (active tab):
    Title: Home | ServiceNow
    URL: https://dev255323.service-now.com/now/nav/ui/home

## AXTree:
Note: [bid] is the unique alpha-numeric identifier at the beginning of lines for each element in the AXTree. Always use bid to refer to
elements in your actions.

Note: You can only interact with visible elements. If the "visible" tag is not
present, the element is not visible on the page.

RootWebArea 'Classic | Unified Navigation App | ServiceNow'
        [47] generic, live='assertive', atomic, relevant='additions text'
        [48] generic, live='polite', atomic, relevant='additions text'

...

# History of interaction with the task:

## step 0

<think>
Looking at the current page, I need to navigate to the "Expense Lines" module in the "Cost" application. First, I need to access the "All"
menu since it's likely where I'll find the Cost application. The "All" menu button is visible with bid "79".
</think>

<action>
click('79')
</action>

## step 1

<think>
I see that the "All" menu is already expanded. I need to find the "Cost" application in the list of applications. Looking through the
AXTree, I can see various applications listed, but I need to search for "Cost" since it's not immediately visible. I can use the search box
at the top of the All menu which has bid "242" and is currently focused.
</think>

<action>
fill('242', 'Cost')
</action>
```

```
## step 2

<think>
I can see that the search box is already filled with "Cost" and the menu is showing filtered results. I can see the "Expense Lines" link
with bid "1145" which is exactly what I need to navigate to. I'll click on this link to access the Expense Lines module.
</think>

<action>
click('1145')
</action>


# Action space:
Note: This action set allows you to interact with your environment. Most of them
are python function executing playwright code. The primary way of referring to
elements in the page is through bid which are specified in your observations.


20 different types of actions are available.

noop(wait_ms: float = 1000)
send_msg_to_user(text: str)
report_infeasible(reason: str)
scroll(delta_x: float, delta_y: float)
fill(bid: str, value: str)
select_option(bid: str, options: str | list[str])
click(bid: str, button: Literal['left', 'middle', 'right'] = 'left', modifiers: list[typing.Literal['Alt', 'Control', 'Shift']] = [])
dblclick(bid: str, button: Literal['left', 'middle', 'right'] = 'left', modifiers: list[typing.Literal['Alt', 'Control', 'Shift']] = [])
hover(bid: str)
press(bid: str, key_comb: str)
focus(bid: str)
clear(bid: str)
drag_and_drop(from_bid: str, to_bid: str)
upload_file(bid: str, file: str | list[str])
tab_close()
tab_focus(index: int)
new_tab()
go_back()
go_forward()
goto(url: str)
Only a single action can be provided at once. Example:
fill('a12', 'example with "quotes"')

Note:
* Some tasks may be game like and may require to interact with the mouse position
in x, y coordinates.
* Some text field might have auto completion. To see it, you have to type a few
characters and wait until next step.
* If you have to cut and paste, don't forget to select the text first.
* Coordinate inside an SVG are relative to it's top left corner.
* Make sure to use bid to identify elements when using commands.
* Interacting with combobox, dropdowns and auto-complete fields can be tricky,
sometimes you need to use select_option, while other times you need to use fill
or click and wait for the reaction of the page.


# Abstract Example

Here is an abstract version of the answer with description of the content of
each tag. Make sure you follow this structure, but replace the content with your
answer:

<think>
Think step by step. If you need to make calculations such as coordinates, write them here. Describe the effect
that your previous action had on the current content of the page.
</think>

<action>
One single action to be executed. You can only use one action at a time.
</action>
```

```
# Concrete Example

Here is a concrete example of how to format your answer.
Make sure to follow the template with proper tags:

<think>
From previous action I tried to set the value of year to "2022",
using select_option, but it doesn't appear to be in the form. It may be a
dynamic dropdown, I will try using click with the bid "a324" and look at the
response from the page.
</think>

<action>
click('a324')
</action>
```