MATHÉMATIQUES ET INFORMATIQUE
Sciences
Université Paris Cité

eurelis
Smart Applications Maker

Université Paris Cité

UFR de MATHÉMATIQUES et INFORMATIQUE

Master MIAGE

Parcours : Valorisation et protection des données de l'entreprise

Automatisation intelligente des tests fonctionnels (E2E) pour les sites web

Génération automatisée de scénarios de test par IA à partir du langage naturel et orchestration multi-agents

Mémoire de Master 2 MIAGE

Auteur :
Nicolas LOISY

Année universitaire 2024-2025

Remerciements

Ce travail a été possible grâce au soutien et aux précieuses contributions de plusieurs personnes que je tiens à remercier.

Un immense merci à Chaima, collègue chercheuse en intelligence artificielle, pour le temps qu'elle m'a consacré et ses explications passionnées sur les concepts avancés de l'IA, des LLM et des RAG.

Merci à Céline, ma tutrice en entreprise, pour son accompagnement bienveillant tout au long de ma montée en compétences professionnelles.

Je suis reconnaissant envers Vincent, PDG de l'entreprise, qui m'a proposé ce sujet de mémoire passionnant.

Merci également à Mr. Sghir, mon tuteur enseignant, pour ses conseils avisés et son suivi durant la rédaction.

Enfin, une pensée spéciale pour mes camarades Clément et Yvonne, avec qui l'entraide et le soutien mutuel ont rendu la rédaction de ce mémoire encore plus enrichissante.

Merci à vous !

Table des matières

Chapitre 1

Introduction

1.1 Contexte et problématique : de l'automatisation traditionnelle à l'IA générative

Les tests fonctionnels des sites web sont impératifs dans le cycle de développement et de déploiement. Ils permettent de vérifier le bon fonctionnement des pages web et des nouvelles fonctionnalités. L'automatisation de ces tests apporte de nombreux avantages par rapport aux tests manuels, particulièrement une plus grande couverture, un temps d'exécution drastiquement diminué, ce qui permet de libérer du temps pour le développeur et de réduire le risque d'erreurs humaines.
Pour l'entreprise, ce gain de temps permet une réduction des coûts de développement et permet de se concentrer sur la qualité du développement ou sur des tests plus approfondis.
Les outils d'automatisation des tests fonctionnels, tels que Selenium [1] (Déf. n°25) ou Cypress [2] (Déf. n°26), ont permis d'automatiser les tests les plus répétitifs et prévisibles.

Les sites web deviennent de plus en plus dynamiques et complexes, ce qui multiplie les interactions possibles sur les pages et rend les tests manuels de plus en plus longs.
Les outils classiques d'automatisation se basent sur la mise en place de scripts de tests qui simulent les interactions d'un utilisateur sur le site.
Ces scripts sont souvent longs à écrire et à mettre à jour, ils ne sont donc pas adaptés à des projets où les nouvelles fonctionnalités sont fréquentes et où les pages changent souvent.
Les tests manuels sont coûteux en ressources humaines, en temps et en argent. Les tests automatiques classiques ont permis de gagner du temps mais restent chers pour les mêmes raisons et sont souvent inadaptés car ils nécessitent une maintenance constante et coûteuse des scripts à chaque modification de l'interface, le gain de temps initial se transforme en une tâche chronophage pour la moindre modification.
Il est donc nécessaire de trouver une solution intelligente et robuste aux changements. L'enjeu pour les entreprises est de diminuer les coûts, d'améliorer la qualité et d'être plus compétitives.

1. Outil d'automatisation des tests web, suite d'outils open-source pour l'automatisation des tests web, pionnier dans le domaine depuis 2004. (Définition n°25)

2. Framework moderne de test end-to-end pour applications web, s'exécutant directement dans le navigateur pour une meilleure expérience développeur. (Définition n°26)

La démocratisation des IA génératives a bouleversé le monde des entreprises. La demande d'intégration de solutions avec de l'intelligence artificielle est forte, même si le besoin n'est pas toujours identifié.
Les grands modèles linguistiques (LLM [3] (Déf. n°4)) ont démontré leur capacité à comprendre et à générer du texte. Aujourd'hui, les grands modèles d'action (LAM [4] (Déf. n°7)) ouvrent une nouvelle voie dans l'automatisation, en permettant de coordonner des actions entre plusieurs applications au sein d'un environnement numérique.

Dans cette course à l'IA [5] (Déf. n°5), le potentiel de cette évolution technologique est sa capacité d'adaptation et de déduction, des qualités humaines qui permettent de réaliser les tests fonctionnels tout en s'adaptant aux petits changements imprévus.

1.2 Hypothèse et objectifs de recherche

J'émets l'hypothèse que l'IA générative peut être utilisée pour automatiser les tests fonctionnels des sites web, en exploitant ses capacités d'adaptation et de compréhension contextuelle pour créer des tests plus intelligents et maintenables.

Ce mémoire vise un double objectif : Réaliser un état de l'art sur l'automatisation des tests fonctionnels web via l'IA générative et les modèles de langage (LLM et LAM). Développer et évaluer une solution concrète qui exploite ces technologies. Il détaillera mon cheminement de pensée et toutes les étapes de mes expérimentations, incluant les réussites comme les contraintes et problèmes rencontrés, pour proposer une application fonctionnelle d'automatisation intelligente des tests.

1.3 Enjeux et motivation de la problématique

La motivation de cette problématique vient de mon expérience en développement web et en intelligence artificielle. J'ai pu observer les limitations des approches traditionnelles d'automatisation des tests et le potentiel inexploité de l'IA générative dans ce domaine. Mon expertise en développement PHP [6] Symfony [7] et Python [8], combinée à ma spécialisation en IA et chatbots, me permet d'explorer cette problématique et de développer une solution concrète et utile.

Le caractère novateur de ce travail réside dans l'application des technologies d'IA générative récentes à l'automatisation des tests fonctionnels web. Contrairement aux outils existants qui s'appuient sur des règles prédéfinies ou de l'apprentissage automatique classique, cette approche exploite la capacité de compréhension contextuelle et d'adaptation

3. Large Language Model : Grand modèle de langage basé sur l'intelligence artificielle, capable de comprendre et de générer du texte de manière contextuelle. (Définition n°4)
4. Large Action Model : Grand modèle d'action capable d'exécuter des séquences d'actions dans des environnements numériques, évolution des LLM vers l'action concrète. (Définition n°7)
5. Intelligence Artificielle : Ensemble de technologies permettant aux machines de simuler l'intelligence humaine pour résoudre des problèmes complexes. (Définition n°5)
6. PHP : Langage de programmation côté serveur utilisé pour le développement web.
7. Symfony : Framework PHP populaire pour le développement d'applications web robustes.
8. Python : Langage de programmation

des modèles génératifs. L'objectif est de créer des tests plus intelligents et maintenables qui s'adaptent automatiquement aux évolutions du code.

Cette recherche entre dans mon projet professionnel de spécialisation entre le développement web et l'intelligence artificielle. Ce mémoire combine un état de l'art approfondi avec une expérimentation pratique. L'objectif est de développer et d'évaluer une solution concrète exploitant les capacités des modèles génératifs pour automatiser la génération de tests fonctionnels web.

Cette problématique concerne principalement les entreprises de services numériques et agences de développement web qui doivent gérer de nombreux projets clients avec des contraintes de temps et de budget serrées. Ces organisations gèrent des applications web complexes nécessitant une couverture de tests très large et une maintenance continue et chronophage.

Les métiers directement impliqués incluent les testeurs, dont le rôle évoluerait vers une supervision de systèmes de test intelligents plutôt que l'écriture manuelle de scripts. Les développeurs web bénéficieraient d'un cycle de développement accéléré grâce à des tests générés automatiquement. Les DevOps [9] (Déf. n°16) pourraient intégrer ces solutions dans leurs pipelines CI/CD [10] (Déf. n°15) pour améliorer la qualité des déploiements. Les product owners et chefs de projet verraient leurs délais de livraison réduits tout en maintenant un niveau de qualité élevé.

1.4 Contexte professionnel

Je réalise mon M2 en alternance au sein d'une entreprise spécialisée dans le développement de solutions web et mobiles. Cette organisation accompagne des clients variés dans la conception et le développement d'applications web, allant des sites e-commerce aux plateformes métier complexes. L'entreprise compte quelques développeurs répartis en équipes projet, utilisant principalement PHP Symfony, React [11], Vue.js [12], Python (Déf. n°22), et diverses solutions mobiles.

L'équipe technique dans laquelle j'évolue est composée de développeurs back-end, de spécialistes front-end et d'un chef de projet. L'entreprise fait face aux défis typiques : manque de temps et de budget sur les projets, nécessité de maintenir la qualité tout en respectant les délais, et besoins croissants en termes de tests automatisés.

Mes missions en entreprise sont autour du développement d'applications web en PHP Symfony et de la conception de solutions d'intelligence artificielle, notamment des chatbots et systèmes conversationnels en Python. Je participe à l'évaluation et à l'intégration de nouvelles technologies, particulièrement dans le domaine de l'IA générative.

9. Méthodologie combinant développement logiciel (Dev) et opérations informatiques (Ops) pour accélérer la livraison de logiciels de qualité. (Définition n°16)

10. Continuous Integration/Continuous Deployment : Pratiques de développement automatisant l'intégration du code et le déploiement des applications. (Définition n°15)

11. React : Bibliothèque JavaScript développée par Facebook pour créer des interfaces utilisateur interactives.

12. Vue.js : Framework JavaScript progressif pour construire des interfaces utilisateur.

Cette expérience en entreprise est directement liée à ma problématique de mémoire. Je constate au quotidien les limitations des approches actuelles de test sur les projets clients. Cette position me permet d'expérimenter avec les technologies d'IA générative sur des cas d'usage réels et de valider mes hypothèses dans un environnement de production. Je peux ainsi développer une solution pratique qui répond aux besoins concrets que j'observe sur le terrain. L'originalité de cette approche réside dans l'utilisation de l'IA générative non seulement pour exécuter des tests mais surtout pour les concevoir, les adapter et les maintenir de manière autonome. Cette recherche a pour ambition de démontrer qu'une solution basée sur les LLM (Déf. n°4) peut révolutionner l'automatisation des tests en réduisant significativement les coûts de développement et de maintenance tout en améliorant la robustesse face aux changements d'interface.

1.5 Plan du mémoire

Pour répondre à la problématique sur l'intégration de l'IA générative dans l'automatisation des tests fonctionnels web, ce mémoire est structuré en quatre parties.

La première partie établit un état de l'art sur l'automatisation des tests à l'ère de l'intelligence artificielle. Elle analyse l'évolution des méthodes d'automatisation des tests fonctionnels web, des approches traditionnelles comme Selenium (Déf. n°25) et Cypress (Déf. n°26) jusqu'aux premières applications de l'intelligence artificielle. Les capacités émergentes des grands modèles linguistiques et des grands modèles d'action sont examinées.

La deuxième partie présente la conception et le développement d'une solution d'automatisation par IA générative. Elle détaille l'architecture, les technologies utilisées (Python, modèles d'IA générative, intégration avec les outils de test existants), et l'approche de développement. Cette partie explicite les hypothèses de travail, justifie les choix techniques, et démontre comment la solution se différencie des approches existantes par sa capacité d'adaptation et sa facilité de maintenance.

La troisième partie étudie l'application de la solution dans une entreprise de développement web. Elle analyse les besoins spécifiques de ce type d'organisation, présente une stratégie d'implémentation incluant les aspects de gestion de projet, de formation des équipes, et d'intégration dans l'environnement technique existant.

La quatrième partie évalue les performances de la solution (temps de développement, taux de succès, robustesse). Les limites identifiées, les avantages confirmés, et les perspectives d'évolution de cette approche y sont discutés. Cette évaluation critique permet de valider l'hypothèse initiale de ce mémoire et de proposer des recommandations pour les entreprises souhaitant adopter ma solution.

Chapitre 2

État de l'art - L'automatisation intelligente des tests fonctionnels web

Le développement web s'accélère, d'autant plus avec l'arrivée des IA, les tests fonctionnels, étape obligatoire du cycle de développement et de mise en production, restent chronophages. L'automatisation a permis de réduire le nombre de tests manuels, ses outils montrent aujourd'hui des limites face à la complexité grandissante des applications web. Ce chapitre est un travail de recherche sur l'évolution des pratiques d'automatisation des tests, depuis les premières solutions jusqu'aux solutions intégrant des formes d'intelligence jusqu'à l'IA. Il comprend l'analyse des forces et faiblesses des outils actuels, et identifie les possibilités d'amélioration qui vont me guider dans le développement de solutions plus intelligentes.

2.1 Fondements de l'automatisation des tests fonctionnels web

2.1.1 Du test manuel à l'automatisation : une évolution nécessaire

Les tests web ont évolué avec le temps. À l'origine, les tests étaient entièrement manuels : un testeur parcourait chaque fonctionnalité, cliquait sur chaque bouton, remplissait chaque formulaire. Cette façon de faire s'est rapidement montrée inadaptée pour les projets. Les problèmes des tests manuels comprennent le temps d'exécution, le risque d'erreur humaine, la difficulté à maintenir une bonne couverture du site web, et un coût élevé. Face à ces contraintes, l'automatisation s'est logiquement imposée comme un besoin économique et qualitatif.

L'arrivée de Selenium (Déf. n°25) en 2004, créé par Jason Huggins, a permis l'automatisation des tests. Ce framework [1] (Déf. n°23) open-source a démocratisé l'automatisation des tests en permettant aux développeurs de scripter les interactions utilisateur. L'évolution s'est poursuivie avec l'intégration des tests dans des pipelines CI/CD (Déf.

1. Structure logicielle réutilisable fournissant des fonctionnalités de base pour développer des applications dans un domaine spécifique. (Définition n°23)

n°15), transformant l'automatisation des tests d'un gain de temps en une étape nécessaire pour toute livraison.

2.1.2 Vue d'ensemble des outils d'automatisation actuels

Il existe aujourd'hui plusieurs outils d'automatisation des tests web, chacun avec ses spécificités et ses compromis.

Selenium : Le plus connu

Selenium reste le pionnier de l'automatisation. Sa force vient de sa grande compatibilité : support de nombreux langages (Java [2], Python, C# [3], JavaScript (Déf. n°21)), compatibilité avec tous les gros navigateurs, et un outil développé et amélioré sur plus de 20 ans. Cette polyvalence vient de son fonctionnement basé sur la technologie WebDriver [4] (Déf. n°24) qui permet comme un humain de naviguer sur les pages, mais cette dernière génère des latences : chaque commande passe par HTTP [5], un aller-retour est effectué, impactant les performances. Le test est créé par un développeur en écrivant un script dans un langage comme Java, Python, C# ou JavaScript, en utilisant WebDriver pour contrôler le navigateur et simuler les actions d'un utilisateur.

Playwright : la modernité Microsoft

Arrivé en 2020, Playwright est une nouvelle génération d'outils. Développé par l'équipe qui avait créé Puppeteer chez Google (un outil similaire), il exploite directement le Chrome DevTools Protocol (CDP [6]) pour communiquer avec les navigateurs. Cette technologie retire les intermédiaires et permet de grandement améliorer les performances et dépassant Selenium (Déf. n°25) pour le même test. Playwright propose également des fonctionnalités comme l'attente automatique et la parallélisation, réduisant la fragilité des tests [7] (Déf. n°2). Les tests peuvent être fragiles car ils dépendent souvent de détails spécifiques de l'interface utilisateur ou du timing, les rendant sensibles aux moindres changements et pouvant échouer malgré qu'il n'y ait pas de réelle erreur. Tout comme Selenium, le test est créé en écrivant un script en JavaScript, TypeScript [8], Python, Java ou .NET [9], en utilisant l'API [10] (Déf. n°17) Playwright pour simuler les actions d'un utilisateur et interagir directement avec le navigateur via le DevTools Protocol.

2. Java : Langage de programmation orienté objet, portable et largement utilisé en entreprise.
3. C# : Langage de programmation orienté objet développé par Microsoft pour la plateforme .NET.
4. Interface standardisée (W3C) permettant aux programmes d'automatiser et de contrôler les navigateurs web de manière programmatique. (Définition n°24)
5. HTTP : HyperText Transfer Protocol, protocole de communication utilisé pour transférer des données sur le web.
6. CDP : Chrome DevTools Protocol, voir définition n°41.
7. Tendance des tests automatisés à échouer lors de changements mineurs d'interface, nécessitant des interventions manuelles fréquentes pour maintenir leur fonctionnement. (Définition n°2)
8. TypeScript : Superset de JavaScript développé par Microsoft, ajoutant un typage statique au langage.
9. .NET : Plateforme de développement multiplateforme créée par Microsoft.
10. Application Programming Interface : Interface de programmation permettant à différents logiciels de communiquer entre eux de manière standardisée. (Définition n°17)

Cypress : la simplicité pour les développeurs

Cypress a fortement amélioré l'expérience développeur en s'exécutant directement dans le navigateur. Cette architecture offre un accès temps réel au DOM [11] (Déf. n°1), la visualisation de la page et une interface de débogage. Pour la rapidité d'exécution, Cypress se positionne entre Selenium (Déf. n°25) et Playwright, mais se démarque sur la facilité d'utilisation. Cypress permet soit d'écrire manuellement des tests au format JavaScript (Déf. n°21) et une logique déclarative des étapes du test, soit d'enregistrer les actions via des outils comme Cypress Studio. Cependant, l'enregistrement des actions avec Cypress Studio est pour l'instant peu fiable et ne permet d'enregistrer que des actions simples. L'écriture manuelle reste fortement utilisée et recommandée pour des tests solides.

TestCafe : la version moderne de Selenium

TestCafe est un outil d'automatisation des tests web utilisant un proxy pour interagir avec les navigateurs, éliminant le besoin de pilotes spécifiques, contrairement à Selenium. Cela simplifie l'installation et assure une isolation complète des tests, offrant une grande stabilité, même pour les applications complexes. Les tests sont écrits en JavaScript ou TypeScript, rendant l'outil accessible et efficace pour les développeurs, avec une syntaxe simplifiée par rapport à celle de Selenium.

Outil	**Architecture**	**Performance**	**Points forts**	**Limitations**
Selenium	WebDriver/HTTP	Lent	Solution la plus utilisée, compatibilité	Lenteur, configuration complexe
Playwright	CDP direct	Rapide	Performance, modernité	Communauté plus petite
Cypress	Dans le navigateur	Rapide	Expérience développeur	Limité à Chromium
TestCafe	Proxy	Variable	Stabilité, isolation	Moins de flexibilité

2.1.3 Patterns et architectures : tentatives d'amélioration de la maintenabilité

Le point commun de tous ces outils est qu'ils nécessitent le développement de scripts de test. Bien que certains outils, comme Cypress avec son Cypress Studio, permettent d'enregistrer des actions pour simplifier partiellement la création de ces tests, la maintenance des scripts reste fastidieuse. En effet, même une petite modification sur un site web peut nécessiter de mettre à jour tout ou partie du script de test pour qu'il continue de fonctionner correctement.

Pour atténuer ce problème, la communauté a mis au point plusieurs bonnes pratiques visant à améliorer la robustesse et la réutilisabilité des scripts de test. Ces pratiques aident à rendre les tests plus faciles à maintenir et à adapter aux changements du site web.

11. Document Object Model : Représentation structurée d'une page web permettant aux scripts d'accéder et de modifier le contenu, la structure et le style des documents web. (Définition n°1)

Le Page Object Model : encapsuler pour une meilleure réutilisabilité

Le Page Object Model (POM [12] (Déf. n°38)) propose d'encapsuler les éléments et interactions de chaque page dans des classes dédiées. Cette approche a plusieurs avantages :

- Centralisation des modifications : un changement d'interface n'impacte qu'une seule classe
- Réutilisabilité du code : les méthodes peuvent servir dans plusieurs tests
- Lisibilité améliorée : les tests deviennent plus expressifs

Malgré ces bénéfices, le POM reste une solution palliative qui ne résout pas fondamentalement le problème de fragilité des tests (Déf. n°2).

Behavior-Driven Development : rapprocher métier et technique

Le Behavior-Driven Development [13] (BDD (Déf. n°36)) est une nouvelle approche de conception des tests. Le BDD se base sur les mêmes principes que le Test-Driven Development [14] (TDD (Déf. n°37)) en se concentrant sur le comportement attendu du logiciel, ce que le logiciel est censé faire du point de vue de l'utilisateur, plutôt que sur son implémentation technique.

On commence par définir les comportements attendus d'un utilisateur sur un site web en utilisant un langage simple et structuré, comme Gherkin [15] (Déf. n°35). Ce langage permet de décrire clairement les fonctionnalités d'une application sous forme de scénarios lisibles par tous, qu'ils soient techniques ou non.

12. POM : Page Object Model, voir définition n°38.

13. Méthodologie de développement centrée sur le comportement attendu du logiciel, favorisant la collaboration équipe. (Définition n°36)

14. Pratique de développement où les tests sont écrits avant le code de production, guidant la conception. (Définition n°37)

15. Langage de spécification lisible utilisant des mots-clés simples (Étant donné, Quand, Alors) pour décrire le comportement logiciel. (Définition n°35)

Par exemple :

```
Scenario : Connexion reussie
  Etant donne que je suis sur la page de connexion
  Quand je saisis des identifiants valides
  Et que je clique sur "Se connecter"
  Alors je suis redirige vers le tableau de bord
```

Le Behavior-Driven Development (BDD) est une méthode qui favorise la collaboration entre les équipes techniques et métiers dans le développement du site.

Avantages :

- Communication simplifiée : Les responsables produit, les testeurs et les développeurs utilisent un langage commun, ce qui facilite la compréhension et la collaboration.
- Documentation : Les spécifications Gherkin servent de documentation toujours à jour, comprenant les dernières spécifications.
- Traçabilité : Le BDD établit un lien direct entre les exigences métiers et les tests exécutés, ce qui permet le suivi des fonctionnalités.

Le processus BDD suit généralement un cycle comprenant les étapes suivantes :

- Découverte : Compréhension et définition des comportements attendus du site web.
- Formulation : Rédaction des scénarios de test en utilisant la syntaxe Gherkin (Déf. n°35).
- Automatisation : Implémentation des "step definitions" qui lient les scénarios aux tests automatisés.
- Exécution : Validation des comportements du logiciel par l'exécution des tests automatisés.

Plusieurs outils soutiennent l'approche BDD, chacun adapté à différents environnements de développement :

- Cucumber : Outil de référence, compatible avec de nombreux langages de programmation tels que Java, JavaScript, Ruby et Python.
- SpecFlow : Spécifiquement conçu pour les projets .NET.
- Behave : Utilisé principalement pour les projets Python.
- Jasmine : Adapté pour les projets JavaScript.

Malgré ses nombreux avantages, le BDD présente certaines limites et défis :

- Effort de maintenance : Chaque scénario Gherkin nécessite une implémentation technique manuelle, ce qui peut doubler l'effort de maintenance.
- Courbe d'apprentissage : Les équipes doivent maîtriser à la fois le langage Gherkin et les outils techniques associés.
- Complexité cachée : La simplicité apparente de la syntaxe naturelle peut masquer la complexité technique.

En pratique, malgré qu'il puisse grandement améliorer la collaboration ainsi que la qualité des logiciels lorsqu'il est bien appliqué, le BDD peut être difficile à maintenir sur le long terme. Cependant, de nombreuses équipes rencontrent en réalité des difficultés à maintenir le BDD après plusieurs mois. Souvent, elles n'arrivent pas à fournir la cohérence nécessaire entre les spécifications et l'implémentation en raison du double effort.

2.1.4 Limites persistantes de l'automatisation traditionnelle

Malgré une vingtaine d'années d'évolution, l'automatisation traditionnelle a des limites fondamentales qui impactent directement le cycle de développement des projets.

Le piège de la maintenance

Le principal problème de l'automatisation traditionnelle réside dans sa rigidité. Les scripts de test sont directement liés à la structure HTML [16] (Déf. n°19) des pages. Un simple changement d'identifiant ou de classe CSS [17] (Déf. n°20) peut faire échouer des dizaines de tests, transformant chaque évolution d'interface en corvée de mise à jour. Ce point crée un élément de réflexion central : l'automatisation, censée accélérer le développement, devient un ralentissement lors des évolutions des pages web.

Expertise requise

L'automatisation traditionnelle exige des compétences techniques. Écrire des tests robustes nécessite une maîtrise des sélecteurs [18] (Déf. n°12) CSS, de la logique d'attente, de la gestion des éléments dynamiques. Ce point technique est l'une des raisons pour lesquelles ces outils ne sont pas toujours utilisés.

Adaptation limitée aux changements

Les outils actuels ne "comprennent" pas l'intention derrière un test. Ils exécutent mécaniquement une séquence d'actions sans pouvoir s'adapter aux variations mineures d'interface. Cette absence d'intelligence contextuelle génère de nombreux faux positifs et nécessite une supervision humaine constante.

Ces limitations montrent que l'automatisation, initialement conçue pour réduire les coûts, peut devenir une source de surcoût si elle n'est pas correctement maîtrisée. C'est dans ce contexte que l'intelligence artificielle émerge comme une potentielle amélioration très prometteuse.

16. HyperText Markup Language : Langage de balisage standard pour créer et structurer le contenu des pages web. (Définition n°19)

17. Cascading Style Sheets : Langage de style décrivant la présentation visuelle des documents HTML (couleurs, polices, mise en page). (Définition n°20)

18. Moyen technique d'identifier précisément un élément dans une page web (CSS, XPath, ID, classe, etc.) pour l'automatisation des tests. (Définition n°12)

2.2 Émergence de l'IA dans l'automatisation des tests

2.2.1 Premières applications de l'IA : vision et auto-réparation

Avant l'explosion des modèles de langage, l'IA (Déf. n°5) s'est d'abord immiscée dans les tests par des applications ciblées visant à résoudre les problèmes les plus criants de l'automatisation traditionnelle.

Computer Vision : des yeux sans utiliser les sélecteurs

Les premières applications concrètes ont exploité la vision par ordinateur [19] pour identifier les éléments d'interface. Cette approche permet de dépasser les limitations des sélecteurs (Déf. n°12) CSS (Déf. n°20) traditionnels en "voyant" réellement les éléments comme le ferait un utilisateur humain. Des algorithmes comme EfficientDet et DETR (DEtection TRansformer) peuvent désormais détecter automatiquement les boutons, champs de saisie et autres éléments interactifs sans dépendre de leur structure HTML (Déf. n°19).

Cette capacité est particulièrement utile pour les tests visuels, où l'IA peut détecter des régressions que les tests fonctionnels traditionnels manqueraient : éléments mal alignés, polices incorrectes, ou problèmes de responsive design.

Auto-réparation : l'adaptation automatique

L'auto-réparation [20] est une grande avancée, sans doute la plus utile, pour la robustesse des tests. Cette fonctionnalité permet aux scripts de s'adapter automatiquement aux modifications mineures d'interface : changement d'identifiant, déplacement d'élément, modification de structure DOM (Déf. n°1).

Le principe repose sur l'utilisation de multiples stratégies de localisation pour chaque élément. Si le sélecteur (Déf. n°12) principal échoue, l'IA teste automatiquement des alternatives (texte visible, position relative, attributs, etc.) pour retrouver l'élément cible. Cette approche réduit considérablement les échecs de test liés aux évolutions d'interface.

2.2.2 Solutions commerciales intégrant l'IA

Plusieurs solutions d'automatisation enrichies par l'IA sont apparues, chacune avec ses spécificités.

Testim : l'objectif de la réduction de maintenance

Testim propose une approche combinant enregistrement visuel et intelligence artificielle. L'outil fonctionne selon ce principe :

1. Enregistrement initial : L'utilisateur navigue dans l'application via le navigateur, Testim capture automatiquement les interactions

19. Domaine de l'IA permettant aux machines d'interpréter et d'analyser le contenu visuel (images, vidéos). (Définition n°33)

20. Mécanisme automatique de détection et correction des erreurs dans les tests, réduisant la maintenance manuelle. (Définition n°32)

2. Analyse IA : L'algorithme analyse chaque élément et utilise la localisation d'élément la plus pertinente (ID, classe, texte, position relative, attributs visuels)
3. Exécution adaptative : Lors de la relecture, si le localisateur principal échoue, l'IA teste automatiquement les alternatives
4. Apprentissage continu : Le système mémorise les succès/échecs pour optimiser les prochaines sélections

L'interface no-code permet aux non-développeurs de créer des tests en glissant-déposant des actions. Testim vend une réduction de 80% des efforts de maintenance.

Limites :

— Dépendance forte à la plateforme SaaS (pas de version on-premise)
— Coût élevé
— Difficultés avec les applications très dynamiques
— Manque de flexibilité pour les scénarios avancés

Applitools : l'expertise du test visuel

Applitools se concentre principalement sur les tests visuels avec sa technologie "Visual AI".
Le fonctionnement repose sur :

1. Capture de référence : Prise de screenshots de l'application dans différents états
2. Analyse sémantique : L'IA découpe l'image en zones fonctionnelles (boutons, textes, contenus)
3. Comparaison intelligente : Lors des tests suivants, l'IA compare les nouvelles captures avec les références
4. Détection contextuelle : L'algorithme ignore les variations acceptables (contenu dynamique, publicités) tout en détectant les vrais problèmes

L'outil est très bon pour détecter des régressions que les tests fonctionnels manqueraient : décalages de pixels, problèmes de fonts, erreurs de responsive design.

Limites :

— Se concentre principalement sur le visuel, ne remplace pas les tests fonctionnels
— Coût élevé pour les projets avec de nombreuses pages
— Nécessite une phase de prise de références longue pour éviter les faux positifs, un état de référence pour tester la non régression du visuel des pages web.

Functionize : l'autonomie par le NLP

Functionize va plus loin en exploitant le traitement du langage naturel [21] (NLP) pour générer automatiquement des tests à partir de simples descriptions en langage courant. Son fonctionnement se passe en plusieurs étapes :

1. Analyse des interactions : observation des interactions utilisateurs types, captées via un tag JavaScript intégré à l'application

21. Branche de l'IA traitant de l'interaction entre ordinateurs et langage humain naturel. (Définition n°34)

2. Génération automatique : création de tests basée sur les analyses des interactions
3. Exécution adaptative : les tests s'ajustent automatiquement aux évolutions de l'interface des pages web
4. Feedback continu : apprentissage à partir des résultats (réussites, échecs, modifications) pour corriger, adapter les tests

Limites :
— Mise en place et configuration complexes selon les témoignages
— Coût élevé
— Dépendance aux données issues d'environnements réels ou proches de la production
— Résultats parfois imprévisibles, qui nécessitent une supervision humaine pour garantir la validité des tests

Mabl : la tentative de simplicité

Mabl propose une solution d'automatisation des tests basée sur une interface visuelle, avec des fonctionnalités de machine learning et une intégration dans les pipelines CI/CD. L'outil permet de créer des scénarios de test sans codage, en enregistrant les interactions utilisateur une à une et en les adaptant automatiquement aux évolutions de l'interface.

Son fonctionnement se passe en plusieurs étapes :

1. Enregistrement visuel : création des tests par capture des actions utilisateur dans le navigateur
2. Auto-maintenance : adaptation automatique des tests en cas de modifications de l'interface (self-healing)
3. Exécution centralisée : lancement des tests sur différents navigateurs et environnements via l'infrastructure cloud de Mabl
4. Analyse intégrée : détection d'erreurs fonctionnelles, de régressions visuelles et génération de rapports détaillés

Limites :
— Période d'apprentissage longue (plusieurs semaines) pour utiliser efficacement toutes les fonctionnalités avancées de l'outil
— Difficultés à distinguer les "vraies" anomalies des évolutions
— Dépendance à l'environnement cloud de Mabl, limitant l'auto-hébergement
— Coût potentiellement élevé pour les équipes avec un volume important de tests
— Nécessite une bonne structuration des parcours testés pour rester fiable

2.2.3 Potentiel des grands modèles linguistiques

L'arrivée de modèles de langage ou LLM (Déf. n°4) comme GPT et Claude représente une avancée dans l'automatisation. Ces technologies ont des capacités impressionnantes pour comprendre et générer du texte, mais leur utilisation dans le domaine des tests en est encore à ses débuts car plusieurs limites existent comme nous avons pu le voir dans les solutions existantes.

Capacités des LLM pour les tests

Les LLM (Déf. n°4) sont excellents dans l'analyse de texte non structuré, ce qui est particulièrement utile pour interpréter des spécifications fonctionnelles [22] et des scénarios de recettes [23]. Ils peuvent extraire des scénarios de test à partir de documents d'exigences rédigés en langage naturel [24], identifier des cas limites potentiels, et même détecter des incohérences dans les spécifications. Ils peuvent extraire des scénarios de test à partir de documents d'exigences rédigés en langage naturel, identifier des cas limites potentiels, et même détecter des incohérences dans les spécifications.

Plusieurs études académiques ont exploré la génération automatique de code de test par les LLM. Les résultats montrent que les modèles obtiennent de meilleurs résultats lorsqu'ils disposent d'informations contextuelles riches : description détaillée du problème, exemples de code existant, et patterns de test établis. Le prompt engineering [25] (façon de formuler des requêtes pour optimiser les réponses des modèles de langage) devient alors crucial pour optimiser la qualité des tests générés.

Limitations actuelles identifiées

Les LLM présentent des limitations importantes pour les tests :

— **Manque de contexte applicatif** : Ils ne "comprennent" pas réellement le fonctionnement de l'application testée
— **Hallucinations** : Génération de code qui semble correct mais contient des erreurs subtiles
— **Incohérence** : Les résultats peuvent varier en fonction de la manière dont les demandes sont formulées.
— **Coût** : Une utilisation intensive peut devenir très coûteuse.

2.2.4 Émergence des modèles d'action : perspectives théoriques

Les concepts de LAM (Déf. n°7) (Large Action Models) et d'agents autonomes commencent à apparaître dans la littérature scientifique. Ces approches promettent d'aller au-delà de la simple génération de texte pour orchestrer des actions concrètes.

Logique des LAM

Les LAM visent à aller plus loin que les modèles de langage classiques (LLM), leur fonctionnement tend vers l'exécution d'actions dans des environnements réels. Plutôt que de se contenter de générer du code, ces modèles pourraient planifier et exécuter des séquences d'actions, s'adapter aux contraintes et événements rencontrés, et optimiser leur stratégie de réponse ou d'action en fonction des résultats obtenus.

22. Document détaillé décrivant les fonctionnalités attendues d'un système, servant de référence pour le développement et les tests. (Définition n°10)
23. Phase de validation fonctionnelle d'une application où l'on vérifie que le système répond aux exigences spécifiées avant sa mise en production. (Définition n°3)
24. Langage humain utilisé spontanément pour communiquer (français, anglais, etc.), par opposition aux langages formels ou de programmation. (Définition n°11)
25. Art et science d'optimiser les requêtes envoyées aux modèles de langage pour obtenir des réponses plus précises et pertinentes. (Définition n°29)

Recherches en cours

Plusieurs laboratoires explorent ces concepts :
— **Multi-agent** : Coordination de plusieurs agents spécialisés pour accomplir des tâches complexes
— **Planning automatique** : Génération de plans de test adaptatifs

Défis techniques non résolus

Ces approches font face à des défis majeurs :
— **Fiabilité** : Comment garantir que les actions exécutées correspondent aux intentions ?
— **Sécurité** : Quels garde-fous pour éviter les actions pouvant causer des dégâts ?
— **Scalabilité** : Ces approches peuvent-elles passer à l'échelle industrielle ?

Ces technologies restent en expérimentation et leur application pratique aux tests fonctionnels web n'existe pas pour l'instant. Le Google Assistant [26] nouvelle génération avec Gemini [27] intégré dans les Android [28] ressemble à un LAM (Déf. n°7) (Large Action Model) en pratique, même si Google ne l'appelle pas officiellement comme ça pour l'instant.

Un LAM, c'est essentiellement un LLM (Déf. n°4) combiné à un ou plusieurs agents exécutifs [29]. Grâce à cette combinaison, on ne se contente plus de comprendre le langage : on peut agir intelligemment en fonction du contexte. Les LAM (Déf. n°7) ouvrent aussi des possibilités pour les tests fonctionnels E2E [30] d'applications web : il devient possible de décrire un scénario utilisateur en langage naturel (Déf. n°11), et de laisser le système le rejouer automatiquement, sans écrire une seule ligne de code de test ni enregistrer des actions au travers d'une interface.

26. Assistant vocal et intelligence artificielle développée par Google.
27. Modèle d'IA multimodal développé par Google.
28. Système d'exploitation mobile développé par Google, basé sur le noyau Linux et destiné aux appareils mobiles. (Définition n°8)
29. Composant logiciel autonome capable d'effectuer des actions automatisées dans un système, fonctionnant de manière indépendante selon des règles prédéfinies. (Définition n°9)
30. End-to-End : Tests de bout en bout validant le fonctionnement complet d'une application du point de vue utilisateur final. (Définition n°14)

2.3 Analyse critique des solutions existantes

2.3.1 Bilan de l'automatisation traditionnelle vs. assistée par IA

Les différences entre les méthodes traditionnelles et celles qui intègrent l'IA sont bien présentes.

L'un des avantages les plus visibles et importants de l'intégration de l'IA dans les tests est la réduction des efforts de maintenance. Des outils comme Testim, capables de se réparer automatiquement, sont vendus avec l'argument de réduire la maintenance jusqu'à 80%. En pratique, cela signifie que les équipes ne passent plus leurs journées à corriger des tests qui ne fonctionnent plus à cause de petites modifications d'interface. De plus, contrairement aux tests traditionnels qui peuvent échouer au moindre changement d'identifiant CSS, les solutions basées sur l'IA utilisent plusieurs stratégies de localisation des éléments sur les pages dans le but de maintenir le fonctionnement des tests, même lorsque l'interface évolue. Cette résistance renforce la confiance dans les résultats des tests et réduit le nombre de fausses alertes.

2.3.2 Métriques d'évaluation et indicateurs de performance

Pour évaluer les bénéfices de l'IA dans l'automatisation des tests, il est nécessaire de définir des métriques de qualité et de performance.

Métriques de productivité

- **Temps de génération des tests** : l'objectif est le passage de plusieurs heures/-jours pour un test complexe à quelques minutes avec l'IA générative
- **Temps d'exécution**
- **Couverture de test** : amélioration de la couverture des tests avec le temps gagné avec les autres points ou grâce à la génération automatique ou semi-automatique de nouveau cas et scénarios de tests alternatifs

Métriques de qualité

- **Taux de faux positifs** : réduction grâce à l'intelligence contextuelle des LLM qui distingue les vrais problèmes des petites variations des sites
- **Stabilité des tests** : amélioration avec l'auto-réparation et l'adaptation automatique

Métriques économiques

- **Coût de maintenance**
- **Temps de mise sur le marché** : accélération des cycles de livraison grâce à des tests plus rapides et fiables

2.3.3 Opportunités d'amélioration avec l'IA générative

En regardant les limites actuelles, on peut voir que les tests automatisés avec de l'IA ont encore une large marge de progression. L'une des pistes les plus prometteuses vient

de l'automatisation via des interfaces en langage naturel. En remplaçant les langages de script ou de programmation par une interface conversationnelle, l'IA générative pourrait rendre la création de scénarios de test accessible aux non-développeurs. Les product owners, testeurs fonctionnels et développeurs seraient ainsi capables de formuler directement leurs cas de test, sans dépendre de spécifications techniques et de développement. Cela permettrait un cycle de livraison plus rapide, et une accélération et automatisation des étapes de validation fonctionnelle du site web.

2.4 Bilan et enjeux futurs

L'automatisation traditionnelle, malgré ses apports, atteint ses limites face à la complexité des sites web. L'intelligence artificielle peut accélérer et changer la donne. D'abord utilisée pour certains points (vision, auto-correction), elle démontre aujourd'hui avec les modèles de langage un grand potentiel : compréhension du contexte ou encore ajustement automatique aux changements.

Les défis restent importants : qualité et validité des tests, intégration sans perdre de temps, accessibilité économique. Ces contraintes justifient le développement de nouvelles façons de faire utilisant l'IA générative.

C'est l'objectif de ce mémoire : concevoir une solution d'automatisation des tests fonctionnels qui s'appuie sur les modèles génératifs pour offrir une approche plus intelligente, adaptative et accessible du test web automatisé.

Chapitre 3

Conception et développement d'une solution d'automatisation par IA générative

3.1 Architecture et approche

3.1.1 Vision globale : vers une automatisation intelligente end-to-end

Automatiser les tests fonctionnels reste un véritable casse-tête depuis toujours. Les outils classiques reposent sur des scripts fixes et sur des sélecteurs (Déf. n°12), c'est-à-dire des identifiants utilisés pour repérer un élément dans l'interface (par exemple le bouton "Valider" ou le champ de recherche). Le problème, c'est que ces sélecteurs sont souvent fragiles, au moindre changement de nom, de position ou de style d'un élément, ils ne correspondent plus. Les tests se cassent, et les équipes de développeurs passent un temps précieux à les réparer, rendant la maintenance coûteuse.

J'ai conçu une approche différente : utiliser l'intelligence artificielle générative pour créer des tests qui s'adaptent automatiquement aux changements. Au lieu de scripts rigides, mon système comprend ce que vous voulez tester et s'adapte quand l'interface évolue.

J'ai nommé le projet TA-Composer ou TAC, le nom vient :
- TA : Test Automation (Automatisation des Tests)
- Composer : Composer/Orchestrer des tests de manière intelligente (Compositeur intelligent de tests)

L'architecture globale repose sur trois couches principales :
- **Couche Intelligence** : C'est le cerveau du système.
- **Couche Orchestration** : L'orchestrateur central coordonne tous les éléments. Il distribue les tâches, supervise l'exécution, et s'assure que chaque composant travaille de manière harmonieuse.
- **Couche Agents Spécialisés** : Chaque agent a une mission précise.

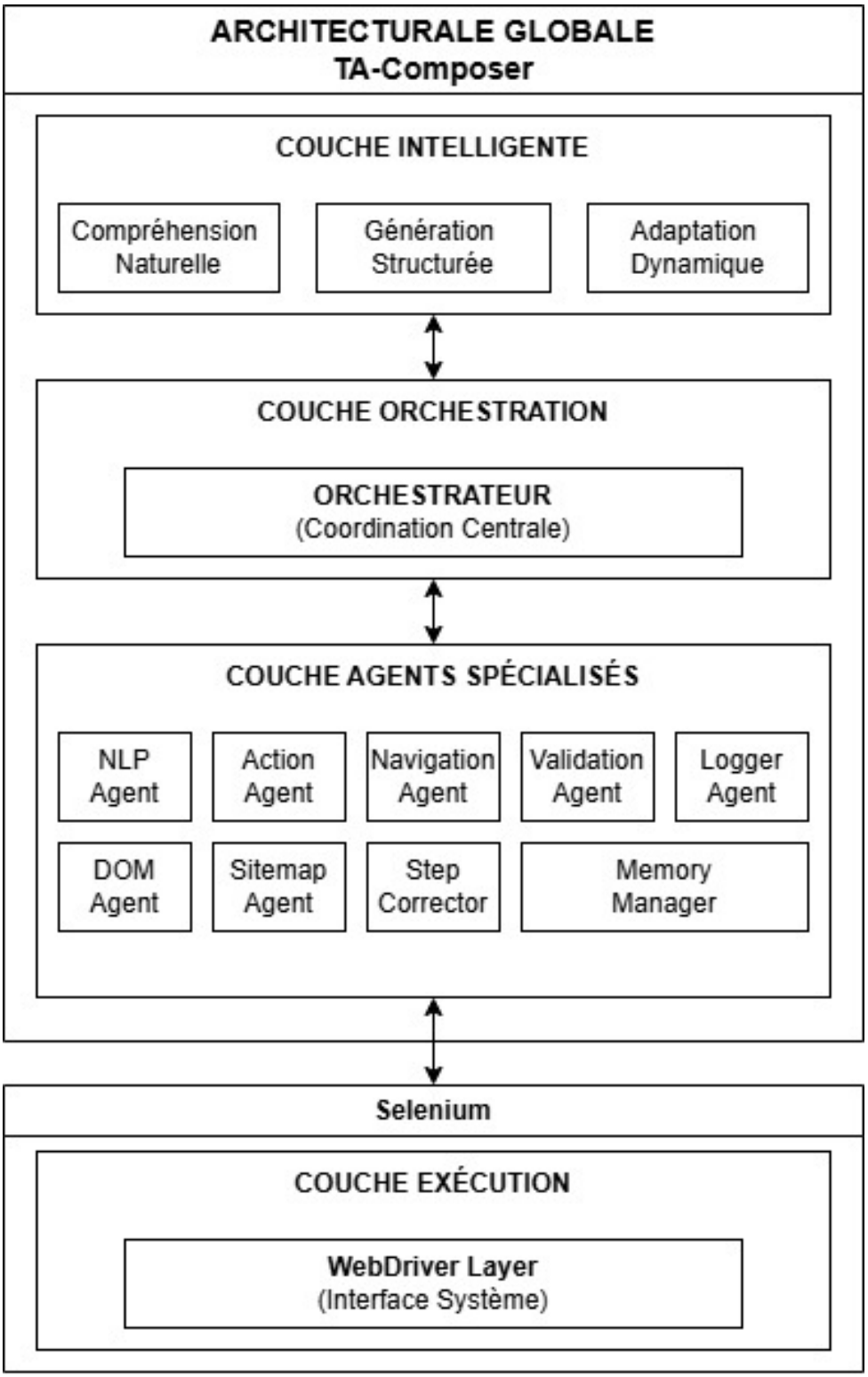

FIGURE 3.1 – Architecture TA-Composer

3.1.2 Fondements théoriques et hypothèses de recherche

L'approche se base sur trois hypothèses, chacune reposant sur les développements récents en intelligence artificielle :

Hypothèse 1 : Capacité de compréhension sémantique des LLM

Les modèles de langage de grande taille (Large Language Models) comme GPT, Claude, Llama ou DeepSeek possèdent des capacités remarquables dans l'interprétation des spécifications fonctionnelles (Déf. n°10) exprimées en langage naturel (Déf. n°11) et peuvent les transformer en séquences d'actions structurées et cohérentes. Une action correspond à une étape précise du scénario de test, telle que l'ouverture d'une page, la saisie d'un champ ou la validation d'un formulaire.

Cette hypothèse repose sur l'utilisation de techniques appelées "structured outputs [1]", qui obligent le modèle de langage à répondre selon un format précis. Un modèle de langage produit en général un texte non structuré, difficile à utiliser tel quel. Grâce à ces techniques, les réponses sont organisées et prêtes à être exploitées directement.

Hypothèse 2 : Résilience par adaptation intelligente

L'intelligence artificielle peut générer des tests suffisamment robustes pour s'adapter automatiquement aux changements mineurs d'interface, réduisant ainsi significativement les coûts de maintenance.

Le terme scientifique qui commence à émerger est "self-healing [2] test automation". L'analyse contextuelle du DOM (Déf. n°1) couplée à un LLM (Déf. n°4) permet d'identifier des éléments équivalents lors de modifications d'interface.

L'implémentation d'un système combinant analyse DOM, raisonnement contextuel IA et génération d'étapes préparatoires donne à la machine ce qui rend actuellement l'humain indispensable dans la correction des scripts de tests fonctionnels.

Hypothèse 3 : Efficacité de l'approche multimodale

L'intégration d'informations textuelles (DOM) et potentiellement visuelles améliore grandement la précision de localisation des éléments et la robustesse des tests. Les avancées en vision par ordinateur (Déf. n°33) et les modèles multimodaux (comme GPT-4V [3]) permettent une meilleure compréhension des interfaces utilisateur. Cela revient à donner la vue à l'ordinateur : l'outil n'analysera plus seulement le code ou le DOM de la page web, mais sera également capable de détecter des problèmes visuels directement sur la page.

1. Technique avancée forçant les modèles d'IA à produire des réponses dans un format structuré prédéfini (JSON, XML, etc.). (Définition n°30)

2. Capacité d'un système à détecter automatiquement ses dysfonctionnements et à appliquer des corrections sans intervention humaine. (Définition n°31)

3. GPT-4V : Version multimodale de GPT-4 capable de traiter à la fois du texte et des images.

3.1.3 Architecture conceptuelle multi-agents

TA-Composer repose sur une architecture multi-agents, où chaque agent remplit une fonction spécifique :

- **L'Orchestrateur** : Coordonne les interactions entre les différents agents.
- **L'Agent NLP** : Transforme les descriptions fonctionnelles en actions techniques exploitables.
- **L'Agent Action** : Exécute les actions sur l'interface utilisateur (clics, saisies, navigation).
- **L'Agent Navigation** : Gère les transitions entre pages ou états de l'application.
- **L'Agent Validation** : Contrôle la bonne exécution des actions et la conformité des résultats.
- **L'Agent Logger** : Enregistre les événements et les résultats pour le suivi et le diagnostic.
- **L'Agent DOM** : Analyse la structure et le contenu des pages web.
- **L'Agent Sitemap** : Explore et cartographie la structure globale de l'application.
- **L'Agent Step Corrector** : Détecte et corrige automatiquement les étapes qui échouent.
- **Le Memory Manager** : Gère le stockage et la récupération des variables partagées entre agents.

Pourquoi cette architecture ?

- **Robustesse** : Si un agent tombe en panne, les autres continuent à fonctionner.
- **Facilité d'évolution** : On peut ajouter de nouveaux agents sans casser toute l'architecture.
- **Spécialisation** : Chaque agent se concentre sur ce qu'il fait le mieux.
- **Maintenance simple** : S'il y a un bug, on sait tout de suite quel agent corriger.

3.1.4 Traitement d'un scénario de test : de l'intention à l'exécution

Quand on demande "Connecter avec admin@test.com", voici ce qui se passe. Le système traite les demandes utilisateur comme "Connecter avec admin@test.com" en suivant un processus en quatre phases :

Phase 1 : Acquisition

Transformation du langage naturel en structures exploitables.
Exemple d'entrée : "Connecter avec admin@test.com"
Processus : Analyse sémantique couplée à l'analyse contextuelle.

Phase 2 : Planification

Décomposition de la demande en étapes séquentielles.

Étapes générées :

1. Navigation → Page Login
2. Action → Saisie Email
3. Action → Saisie Mot de passe
4. Action → Clic Connexion
5. Validation → Vérification Succès

Phase 3 : Exécution adaptative

Exécution avec monitoring en temps réel.
Processus de récupération : Tentative → Détection d'échec → Récupération
Trois issues possibles :
— Succès (exécution réussie)
— Analyse IA (en cas d'échec)
— Nouvel essai (après adaptation)

Phase 4 : APPRENTISSAGE

Apprentissage et optimisation continue
Mécanismes :
— Succès → Sauvegarde des patterns réussis
— Échecs → Amélioration des stratégies
— Contexte → Enrichissement de la base de connaissances
Chaque phase alimente la suivante, créant un cycle d'auto-correction.

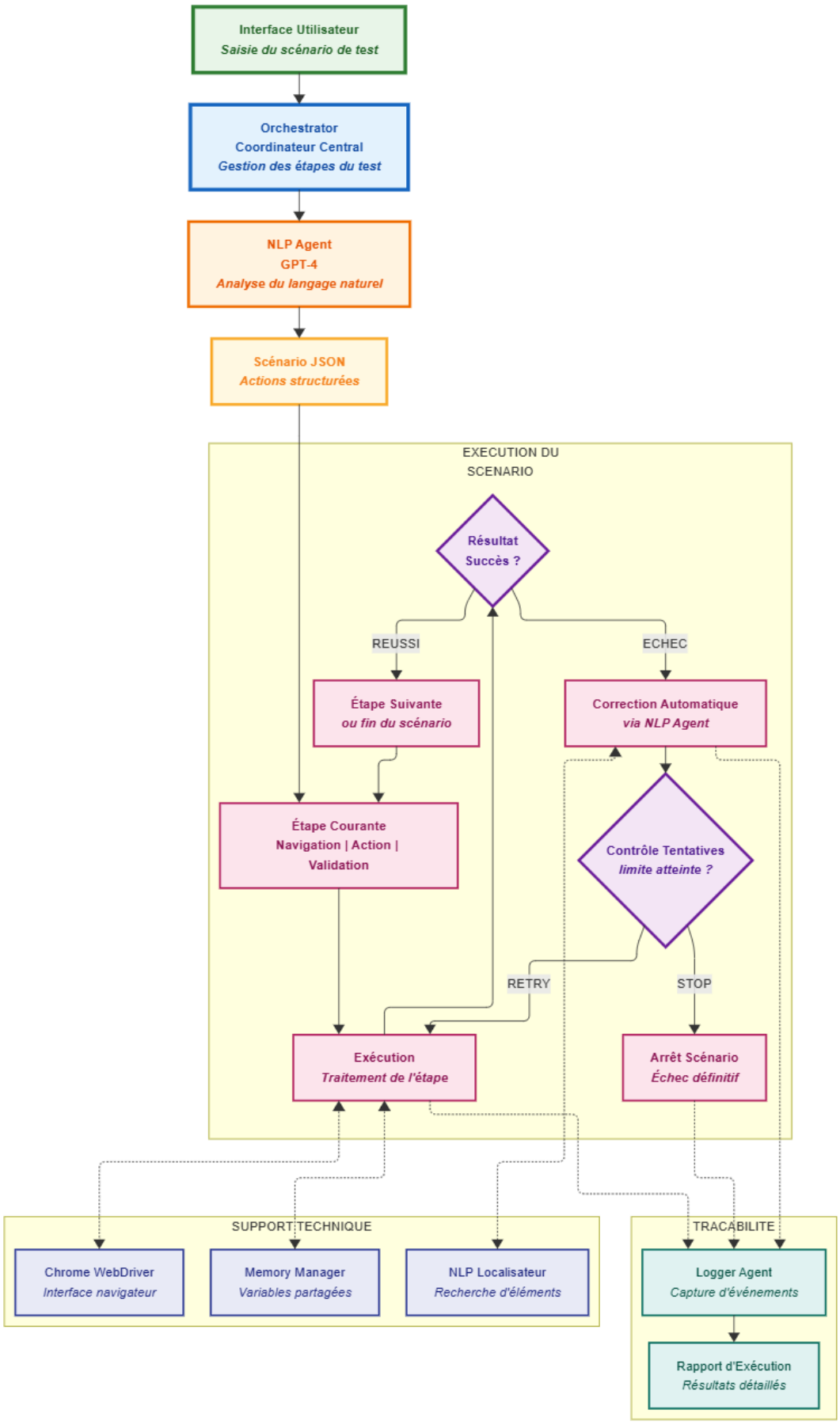

FIGURE 3.2 – Workflow TA-Composer

3.2 Technologies et justifications techniques

3.2.1 Choix des technologies

Justification du langage Python

Python est le langage le plus adapté actuellement pour trois raisons :

Critère 1 : Python bénéficie de l'écosystème le plus complet pour l'intégration d'IA, avec des bibliothèques natives pour tous les principaux fournisseurs de LLM (OpenAI, Anthropic, Hugging Face). Cette richesse réduit la complexité d'intégration et les risques techniques.

Critère 2 : Maturité de l'automatisation web, Selenium WebDriver avec Python possède les modules d'interactions avec les pages web les plus stables et les mieux documentés.

Critère 3 : Productivité de développement, la syntaxe concise de Python facilite l'implémentation de patterns architecturaux comme le Factory Pattern et l'injection de dépendances.

Architecture multi-provider pour l'IA

L'approche utilisée combine les patterns Strategy et Factory pour l'intégration des LLM, assurant une abstraction complète et flexible du fournisseur d'IA.

— **Strategy Pattern** : Permet de sélectionner dynamiquement le provider d'IA (OpenAI, Anthropic, Hugging Face, Ollama, etc.) selon le contexte d'exécution, la disponibilité ou les préférences utilisateur.
— **Factory Pattern** : Centralise la création des instances de LLM, facilitant l'ajout de nouveaux modèles sans modifier le code métier. À partir d'un fichier de config, le bon LLM est créé et utilisé.

Avantages clés :
— **Résilience** : Basculement facilité entre providers en cas d'indisponibilité ou de quota dépassé.
— **Optimisation des coûts** : Choix du provider le plus économique ou performant selon la tâche.
— **Scalabilité** : Ajout de nouveaux modèles ou providers via configuration, sans refonte architecturale du code.
— **Conformité** : Support de solution on-premise (ex : Ollama) pour répondre aux exigences réglementaires.

Cette architecture garantit une intégration de l'IA évolutive, robuste et adaptée aux besoins des entreprises.

3.2.2 Stratégies d'intégration des Large Language Models

Technique des "Structured Outputs"

L'un des grands problèmes de l'utilisation des LLM vient de la garantie de la cohérence des réponses générées. Notre approche utilise les "structured outputs" d'OpenAI (Selon le fournisseur de LLM, le nom peut changer : "response format", "output", etc.), couplés à la validation Pydantic pour garantir la conformité des réponses. Cette technique transforme les réponses de l'IA de simples textes libres en objets Python structurés et validés, évitant les erreurs de format et garantissant la compatibilité avec le reste du système.

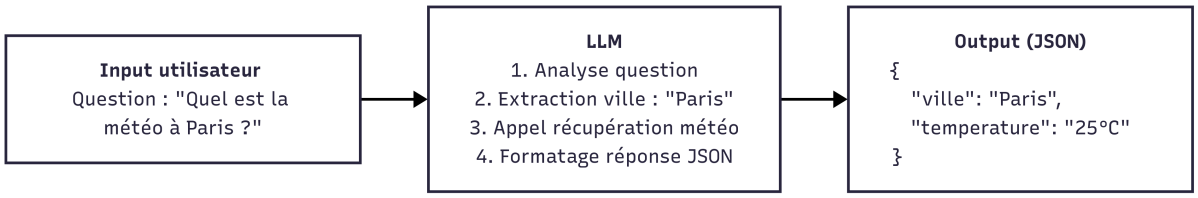

FIGURE 3.3 – Logique des Structured Outputs

Composition du prompt

À partir d'une base de prompt, le système complète dynamiquement le prompt en intégrant automatiquement des informations contextuelles pertinentes (structure DOM, historique de navigation, patterns de succès précédents) afin de générer des étapes de test valides et pertinentes.

3.2.3 Couche d'abstraction Selenium intelligente

Problématique des sélecteurs fragiles

L'automatisation web traditionnelle repose sur des sélecteurs CSS, XPath ou des identifiants DOM pour localiser les éléments sur la page web. Cette façon de faire a un défaut : la fragilité face aux changements. Un bouton change de nom ou d'id et le test n'est plus capable de trouver et d'interagir avec ce dernier.

L'approche de la solution se base sur une couche d'abstraction intelligente qui dépasse cette limitation.

La solution fonctionne sur trois niveaux :
- **Niveau Intention** : L'utilisateur exprime son besoin en langage naturel ("Cliquer sur le bouton de connexion").
- **Niveau Analyse IA** : Le système analyse le DOM de la page et sélectionne la stratégie de localisation la plus appropriée :
  - ID unique
  - Data attributes
  - CSS sémantique
  - XPath structurel
  - Texte visible

— **Niveau Exécution** : L'action est exécutée via WebDriver (WebDriver automatise l'interaction avec le navigateur en simulant clics, saisies et navigation).

Algorithme de correction automatique

Analyse du DOM actuel pour identifier des éléments similaires fonctionnellement à l'élément cible original.

Niveau 1 : Récupération par similarité Analyse du DOM pour trouver des éléments proches (attributs, position, texte) de l'élément cible initial.

Niveau 2 : Génération d'étapes préparatoires Utilisation d'une IA pour détecter et exécuter des actions préalables (fermeture de popups, scroll, changement d'onglet).

Niveau 3 : Réessai avec stratégies alternatives Adaptation de la méthode d'interaction (ex : modifier la méthode de sélection, attendre plus longtemps, simuler une action différente) selon les erreurs détectées.

3.3 Implémentation de la solution

3.3.1 Intégration et utilisation intelligente de Selenium WebDriver

TA-Composer utilise Selenium WebDriver comme moteur d'automatisation web, mais contrairement aux approches traditionnelles, le système l'encapsule dans une couche d'abstraction intelligente qui retire la complexité technique et ajoute des capacités d'adaptation automatique.

WebDriver Factory et gestion des navigateurs

L'instanciation de WebDriver suit le design pattern Factory avec gestion automatique des versions. Le WebDriverFactory utilise webdriver-manager pour télécharger automatiquement la dernière version compatible du ChromeDriver, épargnant les problèmes de compatibilité version navigateur/driver. Le système configure automatiquement les options optimales : taille de fenêtre standard (1920x1080), etc.

Architecture Pattern Command pour les actions Selenium

Chaque interaction Selenium est encapsulée dans une classe d'action spécialisée qui hérite de BaseAction. Cette approche permet de faire des appels Selenium bruts en objets métier intelligents, voici trois exemples :

— **ClickAction** : Encapsule driver.find_element().click() avec détection automatique des nouvelles fenêtres. Après chaque clic, le système mémorise les handles de fenêtres, détecte l'ouverture de nouveaux onglets et bascule automatiquement vers la nouvelle fenêtre, éliminant les erreurs classiques de gestion multi-fenêtres.

— **InputTextAction** : Surcharge driver.find_element().send_keys() avec résolution de variables mémoire. Le système intègre le Memory Manager pour résoudre automatiquement les variables {nom_variable} dans les valeurs saisies, permettant la réutilisation de données entre étapes.

— **WaitAction** : Remplace les time.sleep() statiques par des attentes paramétrables avec feedback utilisateur via l'OutputManager colorisé.

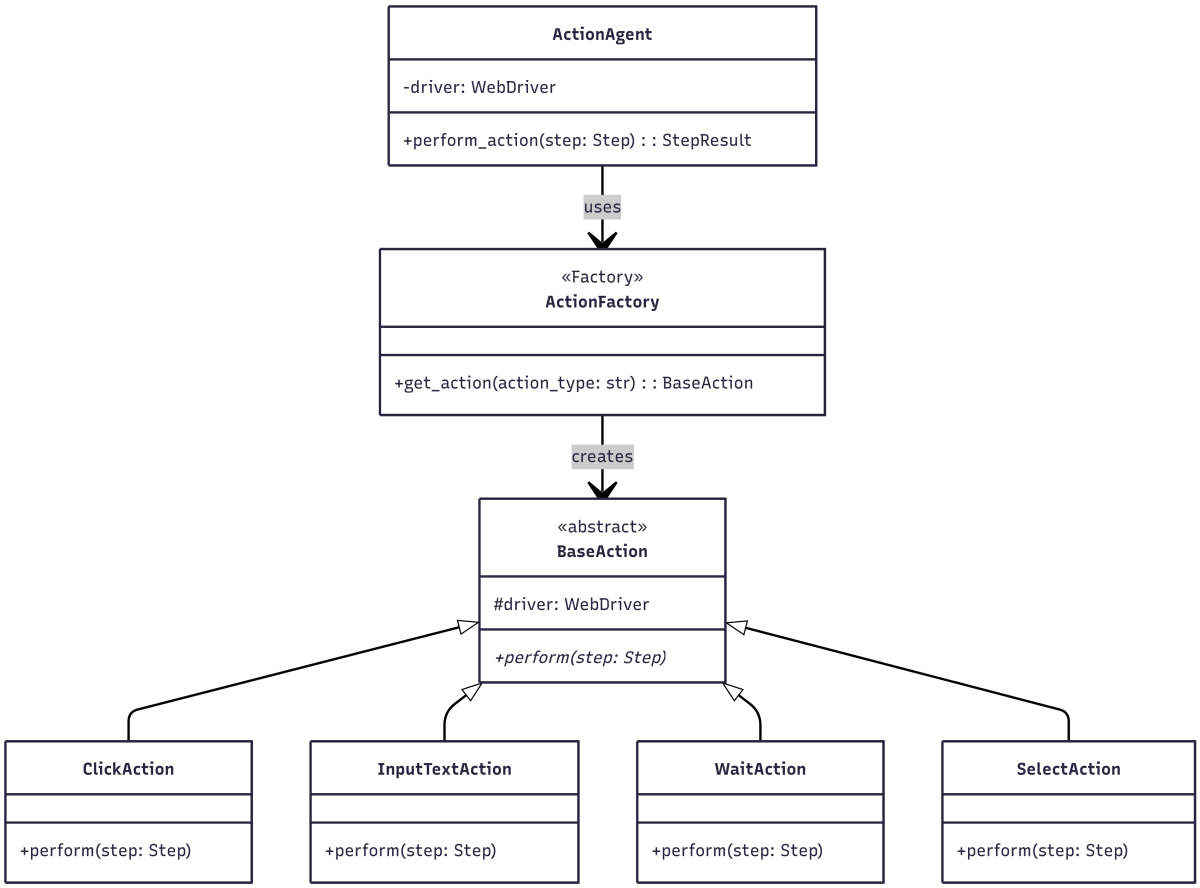

FIGURE 3.4 – Diagramme de classes UML - TA-Composer

Fonctionnement : L'ActionAgent reçoit une demande d'action et demande à l'ActionFactory de créer la bonne action (clic, saisie, attente, etc.), toutes ces actions suivant le même modèle défini par BaseAction pour garantir une compatibilité d'exécution.

Avantages : Cette approche permet d'ajouter facilement de nouveaux types d'actions sans casser l'existant, chaque composant a un rôle bien défini, et toutes les actions fonctionnent de la même manière pour simplifier la maintenance.

Optimisation du contenu pour l'IA via HTMLCleaner

Le module HTMLCleaner est utilisé avant chaque appel au LLM, il nettoie automatiquement le DOM récupéré par Selenium avant transmission à l'IA : suppression des balises script/style/link, élimination des SVG volumineux, retrait des images base64, et formatage pour optimiser la compréhension du DOM par les LLM. Cela permet de réduire la taille du contenu envoyé au LLM et améliore la précision des analyses DOM par l'IA tout en réduisant les coûts.

Gestion intelligente des sélecteurs

Contrairement aux scripts Selenium traditionnels utilisant des sélecteurs fixes, TA-Composer localise les éléments intelligemment avec le LLM. Le système teste automatiquement plusieurs stratégies Selenium (By.ID, By.CLASS_NAME, etc.) selon la stratégie la plus fiable selon l'IA, avec correction automatique en cas d'échec.

Sélecteur	**Description**
By.ID	Localise un élément par son attribut id unique
By.CLASS_NAME	Cible les éléments par leur classe CSS
By.NAME	Localise par l'attribut name du formulaire
By.LINK_TEXT	Trouve un lien par son texte exact affiché
By.PARTIAL_LINK_TEXT	Trouve un lien par une partie de son texte
By.TAG_NAME	Sélectionne par le nom de la balise HTML
By.CSS_SELECTOR	Utilise la syntaxe CSS pour cibler des éléments
By.XPATH	Navigue dans la structure DOM avec des expressions XPath

TABLE 3.1 – Stratégies de sélection Selenium

TA-Composer permet d'encapsuler Selenium et masque les détails techniques aux utilisateurs finaux : gestion automatique des timeouts, gestion des exceptions WebDriver avec classification intelligente des erreurs et attente automatique du chargement des pages. La solution transforme Selenium d'un outil technique complexe en moteur d'exécution de tests complètement transparent contrôlé par l'intelligence artificielle.

Gestion des dépendances et configuration

TA-Composer utilise un fichier de configuration JSON permettant de paramétrer les fournisseurs LLM (OpenAI, Anthropic, Ollama), les options Selenium, et les niveaux de logging. Ce fichier de configuration facilite la personnalisation de l'outil, les déploiements multi-environnement et l'adaptation aux contraintes des entreprises.

3.3.2 Interface utilisateur et workflow d'utilisation

L'application s'exécute via une interface ou une console proposant deux modes distincts :

Mode 1 - Exécution de tests existants : L'utilisateur spécifie le chemin d'un fichier JSON contenant un scénario de test. Le système charge le scénario, l'exécute via l'Orchestrateur, et génère un rapport d'analyse enrichi par l'IA.

Mode 2 - Génération de nouveaux tests : L'utilisateur décrit son besoin en langage naturel ("Se connecter en tant qu'administrateur et créer un nouveau produit"). Le NLPAgent transforme cette description en scénario structuré, le sauvegarde automatiquement, puis l'exécute immédiatement.

Workflow utilisateur

1. Lancement de l'application via python main.py
2. Sélection du mode d'opération
3. Saisie des paramètres (fichier ou description)
4. Exécution automatique avec affichage temps réel
5. Génération du rapport final avec recommandations

L'OutputManager fournit un retour utilisateur avec différents niveaux (DEBUG, INFO, WARNING, ERROR, SUCCESS, CRITICAL). Chaque étape d'exécution est tracée avec son statut, permettant un suivi précis du déroulement des tests.

3.3.3 Formats de réponse structurée (Structured Outputs)

TA-Composer force le LLM à retourner uniquement ces structures de données précises grâce aux Structured Outputs d'OpenAI. Au lieu de recevoir du texte libre, l'IA doit obligatoirement générer des objets Scenario (contenant une liste de Step) ou des Step individuelles avec les champs exacts définis.

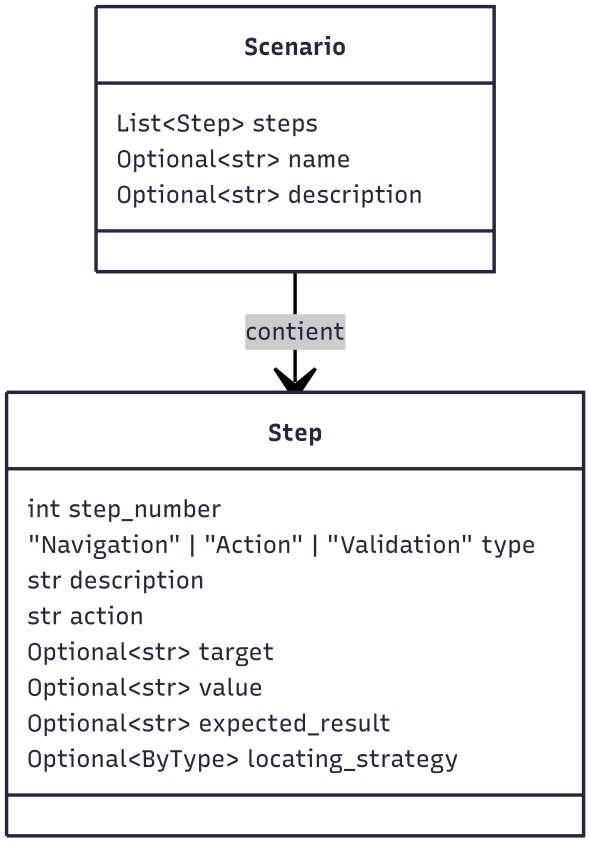

FIGURE 3.5 – Schéma de données : Réponse structurée

Chaque Step contient le type d'action (Navigation/Action/Validation), l'action spécifique à effectuer (click, input_text, etc.) et les paramètres de localisation (target + locating_strategy) permettant d'identifier précisément l'élément sur la page.

Cette contrainte élimine totalement les erreurs de parsing et garantit que chaque réponse IA est immédiatement exploitable par TA-Composer, sans interprétation de la réponse ou conversion supplémentaire.

3.4 Différenciation et Positionnement Innovant

3.4.1 Analyse Comparative avec l'État de l'Art

Limitations des solutions actuelles

Selenium, Cypress et TestCafe partagent les mêmes défauts : maintenance lourde et complexité technique. Ces outils cassent dès qu'une interface change.

Différenciation avec TA-Composer :

Critère	**TA-Composer**	**Selenium, Cypress et TestCafe**
Barrière d'entrée	Langage naturel	Code technique
Adaptabilité	Auto-correction	Scripts figés
Maintenance	Automatique	Manuelle
Apprentissage	Immédiat	Formation technique longue
Architecture	Modulaire	Monolithique

Table 3.2 – Comparaison avec les outils traditionnels

3.4.2 Innovations Théoriques et Pratiques

Innovation 1 : Architecture multi-agents intelligente Avec des agents pour la coordination, la compréhension, la planification et l'exécution.

Innovation 2 : L'utilisateur dit "quoi faire", l'IA détermine "comment le faire". Plus besoin de spécifier chaque action technique. On dit "Se connecter avec admin" et l'IA traduit en une succession d'actions.

Innovation 3 : Logique LAM (Large Action Models) TA-Composer implémente la logique des Large Action Models : l'IA exécute directement les actions au lieu de générer du code. Le système agit comme un agent autonome qui navigue et interagit avec les interfaces.

La solution résout de vrais problèmes : démocratise l'automatisation, accélère la mise en production, améliore la qualité, réduit la maintenance.

Chapitre 4

Validation de la solution

4.1 Méthodologie de test

4.1.1 Critères d'évaluation

La validation de TA-Composer doit vérifier trois points : génération de tests, exécution automatisée, et analyse de performance. La validation de la solution doit évaluer les capacités d'adaptation, la robustesse et l'efficacité de la solution par rapport aux outils de tests traditionnels.

Critères d'évaluation définis

Génération intelligente : Capacité du système à transformer des descriptions en langage naturel en scénarios de test structurés et exécutables. Mesure par le taux de réussite de génération et la cohérence sémantique des étapes produites.

Adaptabilité : Résistance aux changements mineurs d'interface (modification de sélecteurs, déplacements d'éléments) grâce aux mécanismes d'auto-correction. Évalué par le nombre d'interventions manuelles nécessaires lors d'évolutions d'interface.

Robustesse : Capacité à gérer les cas d'échec et à appliquer des stratégies de récupération intelligentes. Mesuré par le taux de succès après correction automatique et le nombre d'étapes supplémentaires générées.

Performance : Temps d'exécution des tests et efficacité des interactions avec les modèles d'IA. Comparaison avec des scripts Selenium traditionnels équivalents.

4.1.2 Sites web sélectionnés et environnements de test

Des sites web réels ayant différentes formes de complexité sont nécessaires pour valider TA-Composer :

Site principal d'expérimentation : nicolasloisy.fr

- **Justification :** Site personnel développé avec des technologies modernes (HTML5, CSS3, JavaScript)
- **Complexité :** Interface dynamique avec projets interactifs (Machine Enigma), navigation multi-pages
- **Avantages :** Contrôle total sur l'environnement, possibilité de modifier l'interface pour tester l'adaptabilité

Applications web interactives intégrées :

— **Machine Enigma :** Application de chiffrement historique avec formulaires complexes, interactions asynchrones, et validation de résultats
— **Éléments testés :** Configuration de rotors, connexions plugboard, encodage/décodage de messages

Environnement technique :

— **Navigateur :** Chrome (version récente avec ChromeDriver automatiquement mis à jour)
— **Résolution :** 1920x1080 (standard configuré automatiquement)
— **Système :** Windows 10/11, Python 3.8+
— **Infrastructure LLM :** OpenAI GPT-4o-2024-08-06 avec structured outputs

4.2 Cas d'usage et validation pratique

4.2.1 Navigation et parcours utilisateur complexe

Cas d'usage 1 : Parcours de découverte de portfolio

Le scénario de test est un parcours utilisateur sur un site portfolio. L'objectif est de valider la présence d'éléments clés (bouton de téléchargement du CV, projet Enigma) et de tester l'accès à la démonstration du projet Enigma. Ce scénario permet d'évaluer la capacité de TA-Composer à générer et exécuter automatiquement des étapes de test à partir d'une simple description fonctionnelle, tout en s'adaptant aux éventuelles évolutions de l'interface.

Description en langage naturel fournie : "Aller sur nicolasloisy.fr, vérifier que le bouton CV existe, vérifier que le projet Enigma existe, cliquer sur le lien démo du projet Enigma"

Scénario généré automatiquement par TA-Composer :

```
{
  "steps": [
    {
      "step_number": 1,
      "type": "Navigation",
      "description": "Navigate to the website nicolasloisy.fr",
      "target": "https://nicolasloisy.fr",
      "action": "go_to_url"
    },
    {
      "step_number": 2,
      "type": "Validation",
      "description": "Check if the CV button exists on the page",
      "target": "T l charger le CV",
      "locating_strategy": "link text",
      "validation_type": "presence"
    },
    {
```

```
      "step_number": 3,
      "type": "Validation",
      "description": "Check if the Enigma project exists on the page",
      "target": "//h3[text()='Enigma']",
      "locating_strategy": "xpath",
      "validation_type": "presence"
    },
    {
      "step_number": 4,
      "type": "Action",
      "description": "Click on the demo link of the Enigma project",
      "target": "Voir le projet",
      "locating_strategy": "link text",
      "action": "click"
    }
  ]
}
```

Résultat d'exécution :

- **Taux de succès :** 100% (4/4 étapes réussies)
- **Temps d'exécution total :** < 1 minute
- **Adaptations automatiques :** Le système a automatiquement utilisé différentes stratégies de localisation (link text, xpath) selon le contexte de chaque élément

4.2.2 Interactions avec formulaires et gestion d'état

Cas d'usage 2 : Configuration et test de la Machine Enigma

Description en langage naturel : "Sur la machine Enigma, configurer les rotors avec 3, 4, 6, ajouter une connexion BR, encoder BONJOUR, puis réencoder le résultat pour vérifier qu'on obtient BONJOUR"

Complexité du scénario :

- Gestion d'état complexe (configuration des rotors)
- Interactions asynchrones (attente du résultat de l'encodage)
- Utilisation du Memory Manager (stockage et réutilisation de valeurs)
- Validation de logique métier (principe de réversibilité d'Enigma)

Étapes clés générées :

```
{
  "step_number": 14,
  "type": "Action",
  "description": "Store the encoded result text",
  "target": ".result",
  "expected_result": "encoded_result",
  "locating_strategy": "css selector",
  "action": "store_value"
},
{
  "step_number": 15,
  "type": "Action",
  "description": "Enter the stored result in the message field",
  "target": "#message-to-encode",
  "locating_strategy": "css selector",
```

```
  "action": "input_text",
  "value": "{encoded_result}"
}
```

Innovation démontrée : Le système utilise automatiquement le Memory Manager pour stocker la variable encoded_result et la réutiliser dans l'étape suivante via la syntaxe {encoded_result}. Cette capacité n'est pas présente dans les possibilités des outils de test courants qui nécessitent un développement pour gérer ce type de vérification.

4.2.3 Validation de contenu et mécanismes d'auto-correction

Cas d'usage 3 : Robustesse face aux changements d'interface

Scénario de test d'adaptabilité : Modification volontaire des identifiants CSS de la Machine Enigma (changement de l'identifiant d'un champ de la page : de #rotor1 vers #rotor-1-new) pour évaluer les capacités d'auto-correction sur un scénario de test déjà généré.

Comportement observé :

1. **Détection d'échec :** Le système ne parvient pas à localiser le #rotor1
2. **Analyse IA contextuelle :** Le NLPAgent analyse le DOM nettoyé et identifie #rotor-1-new comme élément équivalent
3. **Correction automatique :** L'ActionAgent applique la correction et poursuit l'exécution
4. **Apprentissage :** Le nouveau sélecteur est mémorisé pour les prochaines exécutions
5. **Bilan :** Les changements sont notifiés dans le rapport d'exécution

Métriques de récupération :

— **Temps de détection d'erreur, de l'analyse et de la correction :** <1 minute
— **Taux de succès après plusieurs exécutions de ce test :** 8 fois sur 10

Cette capacité d'auto-correction est une évolution par rapport aux scripts traditionnels qui échoueraient dans cette situation.

4.3 Configuration de la solution

4.3.1 Paramétrage des modèles IA

Configuration LLM optimisée :

```
{
  "llm": {
    "provider": "openai",
    "model": "gpt-4o-2024-08-06",
    "temperature": 0.1,
    "max_tokens": 4000,
    "timeout": 30
  }
}
```

Justifications techniques :

- **Modèle GPT-4o-2024-08-06 :** Modèle performant avec un bon rapport qualité de réponse et coût du token. Ce modèle supporte les "structured outputs".
- **Température 0.1 :** La température du modèle rend la réponse plus déterministe et moins aléatoire.

4.3.2 Intégration et workflow opérationnel

Interface utilisateur :

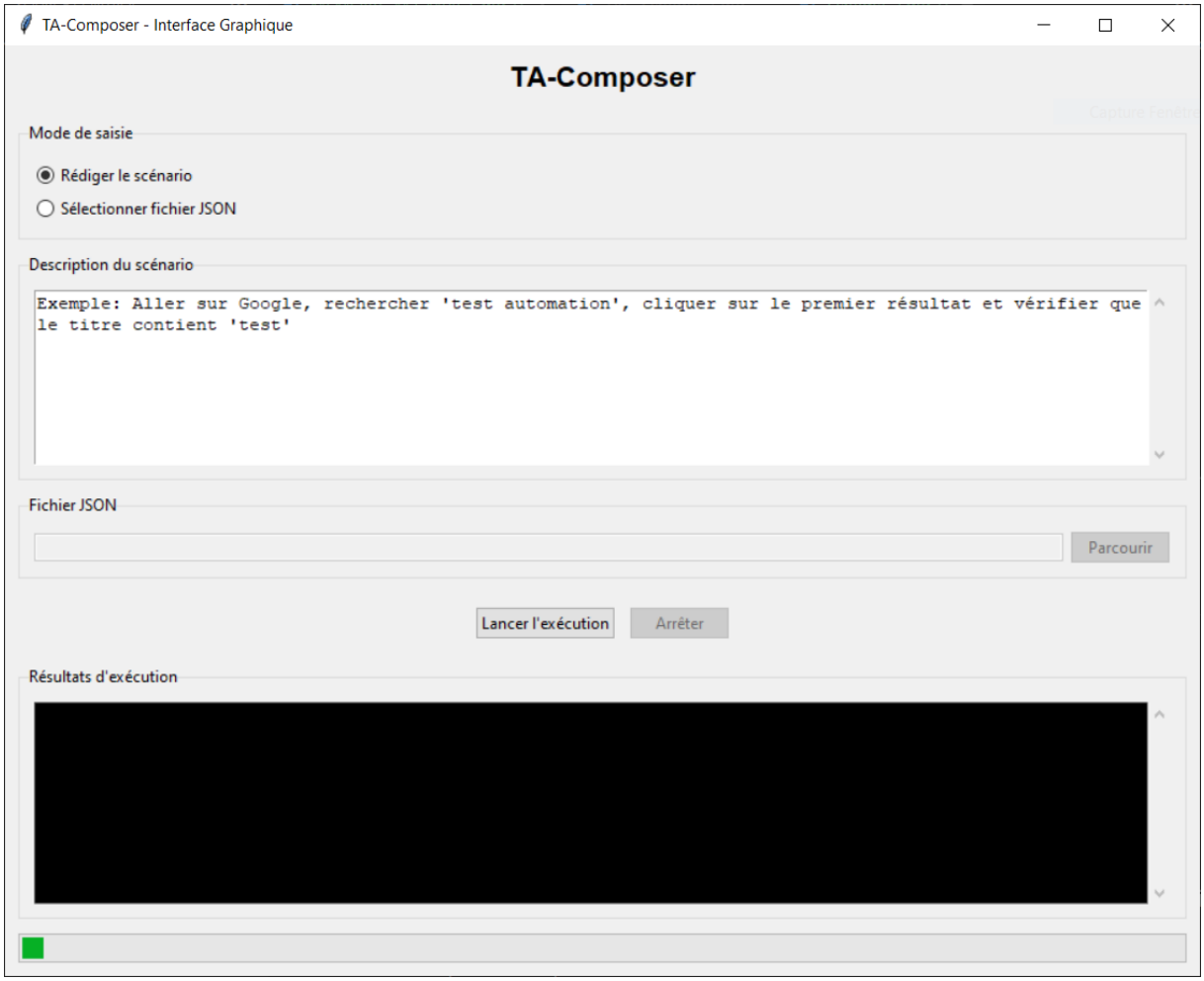

FIGURE 4.1 – TA-Composer : Interface utilisateur

Modes d'utilisation :

Mode 1 - Exécution de tests existants :

```
cd src
python main.py
# Choix : 1 (Executer un test JSON existant)
# Fichier : tests/generated_test_20250805013105.json
```

Mode 2 - Génération avec scénario de test :

```
cd src
python main.py
# Choix : 2 (Generer un test)
# Description : "Tester la machine enigma avec configuration 3,4,6"
```

Workflow d'exécution automatisé

Les fichiers de configuration et les interactions pouvant se faire entièrement par console, TA-Composer peut démarrer très simplement et générer son rapport après son exécution. Son intégration dans les pipelines CI/CD et les workflows DevOps via des scripts d'automatisation est facilitée, permettant d'automatiser les tests après chaque développement.

Monitoring et observabilité

Le système génère automatiquement :
— **Logs d'exécution :** Traçabilité complète dans logs/execution_log.txt
— **Rapports détaillés :** Analyse IA des succès/échecs avec recommandations avec le rapport d'exécution

Les résultats confirment les hypothèses initiales sur les capacités d'adaptation et la réduction des coûts de maintenance par rapport aux approches traditionnelles.

Chapitre 5

Conclusion

Ce mémoire a pour ambition de dépasser les limitations des approches classiques d'automatisation des tests fonctionnels web End-to-End (E2E) en exploitant le potentiel de l'IA générative. Le constat de la rigidité des scripts de test traditionnels et de leur coût de maintenance élevé montre qu'une innovation est nécessaire, l'objectif a été de concevoir et de valider une méthode permettant de générer dynamiquement des scénarios de test à partir du langage naturel, le tout orchestré par une architecture multi-agents comportant une couche intelligente avec IA.

Cette problématique est d'autant plus importante avec l'accélération des cycles de développement et de la complexité croissante des applications web.

5.1 Synthèse de l'approche et validation des hypothèses

Face aux limites persistantes des outils d'automatisation traditionnels :
— Maintenance coûteuse
— Rigidité face aux changements d'interface
— Expertise technique requise

TA-Composer a été conçu pour faire face aux limites des outils de tests traditionnels, une solution innovante qui exploite les capacités des modèles LLM pour améliorer l'approche des tests automatisés.

L'architecture multi-agents développée valide les trois hypothèses initiales. L'hypothèse de compréhension sémantique a été confirmée par la capacité du système à transformer efficacement des descriptions en langage naturel ("Se connecter avec admin@test.com") en séquences d'actions structurées et exécutables. L'hypothèse de résilience s'est vérifiée par les mécanismes d'auto-correction qui permettent au système de s'adapter automatiquement aux modifications mineures d'interface, avec un bon taux de succès sur nos cas de tests.

Les résultats de validation sur un site en ligne dans un environnement réel démontrent l'efficacité de la solution : génération automatique de scénarios de test, exécution avec mécanismes d'auto-correction, et réduction des interventions manuelles.

5.2 Enseignements et contributions scientifiques

Cette recherche, état de l'art et conception de la solution, apporte plusieurs contributions au domaine de l'automatisation des tests.

Sur le plan théorique, nous avons montré la faisabilité d'une implémentation des concepts de Large Action Models (LAM) dans le contexte spécifique des tests web. L'orchestration intelligente d'agents spécialisés reproduit les capacités d'adaptation et de planification qui caractérisent les LAM.

Sur le plan méthodologique, TA-Composer montre qu'on peut rendre l'automatisation des tests accessible à tous, sans compétences techniques. Il suffit de décrire les tests en langage naturel, ce qui permet aux testeurs fonctionnels, product owners et autres membres non techniques de participer facilement aux tests.

5.3 Bénéfices

La solution a plusieurs avantages pour les équipes de développement.

L'accélération du cycle de développement Réduction du temps de création des tests : de plusieurs heures pour un scénario complexe à quelques minutes de description fonctionnelle. Cette accélération libère du temps et donc de l'argent.

L'amélioration de la qualité Les équipes peuvent désormais créer plus facilement des scénarios de test diversifiés.

La libération des contraintes de maintenance Les mécanismes d'adaptation automatique aux évolutions d'interface retirent une tâche chronophage.

5.4 Limites identifiées et défis restants

La dépendance aux modèles de langage externes Avec une utilisation intensive, les coûts peuvent devenir significatifs, particulièrement pour les organisations avec de nombreux tests quotidiens. L'intégration de solutions on-premise comme Ollama dans notre architecture multi-provider répond partiellement à cette préoccupation, mais nécessite un équipement performant pour exécuter un LLM.

La complexité des scénarios très spécialisés TA-Composer fonctionne dans les parcours utilisateur classiques et courts, les scénarios de tests nécessitant une connaissance métier très spécifique ou des interactions avec des systèmes complexes nécessitent une action humaine.

5.5 Perspectives d'évolution et roadmap technique

L'intégration de modèles de vision enrichit l'analyse avec une IA ayant des capacités de vision va permettre d'améliorer la détection d'anomalies visuelles et la localisation d'éléments. Les progrès récents en IA visuelle (GPT-4V, Claude 3.5 Vision) vont faciliter l'intégration.

L'ajout de nouvelles Actions TA-Composer peut évoluer avec l'ajout de nouvelles actions complexes telles que le dépôt de documents ou encore la validation de CAPTCHA avancés (via services tiers ou modèles de vision). On peut également étendre le champ d'action de TA-Composer à des tests d'interactions avec des API. Grâce à son architecture multi-agents et multi-actions, chaque nouvelle action peut être intégrée facilement.

5.6 L'évolution des pratiques

La démocratisation d'un tel outil changera les pratiques de test dans les entreprises.

Les testeurs évolueront vers une fonction de supervision et d'analyse plutôt que de création de tests par développement et d'exécution manuelle.

L'intégration CI/CD de TA-Composer facilite l'adoption de pratiques DevOps, où les tests deviennent une étape facile à mettre en place dans les pipelines de livraison continue.

La démocratisation de l'automatisation va permettre une meilleure collaboration entre les équipes techniques et fonctionnelles.

5.7 Positionnement technologique

Ce mémoire entre dans les nombreuses possibilités qui découlent des avancées récentes en IA générative. L'apparition des concepts de LAM et des agents autonomes va sans doute constituer la prochaine étape après les IA génératives dans le domaine de l'automatisation en général, tous secteurs et supports confondus. TA-Composer est une proposition d'architecture utilisant l'IA, un outil qui a les capacités de dépasser les outils de test traditionnels et de rendre l'automatisation plus simple et rapide.

Annexe A

Bibliographie

A.1 Articles de revues et publications scientifiques

Ali, H. M., Hamza, M. Y., & Rashid, T. A. (2024). A Comprehensive Study on Automated Testing with the Software Lifecycle. *arXiv preprint*. https://arxiv.org/pdf/2405.01608

Barua, R., Ghosh, S. K., & Rahman, A. (2025). The Future of Test Automation : A Comparative Analysis of Selenium vs. AI-Driven Tools. *ResearchGate*.https://www.researchgate.net/publication/392283123_The_Future_of_Test_Automation_A_Comparative_Analysis_of_Selenium_vs_AI-Driven_Tools

Publishing House. (2024). A comparative analysis of web application test automation tools. *Journal of Computer Science and Information*. https://ph.pollub.pl/index.php/jcsi/article/view/7119

ResearchGate. (2024). AI and Machine Learning in Test Automation : Techniques, Challenges, and Future Prospects. https://www.researchgate.net/publication/383692865_AI_and_Machine_Learning_in_Test_Automation_Techniques_Challenges_and_Future_Prospects

Sherifi, B., Slhoub, K., & Nembhard, F. (2025). The Potential of LLMs in Automating Software Testing : From Generation to Reporting. *arXiv preprint*. https://arxiv.org/html/2501.00217v1

Wang, L., Yang, F., Zhang, C., Lu, J., Qian, J., He, S., Zhao, P., Qiao, B., Huang, R., Qin, S., Su, Q., Ye, J., Zhang, Y., Lou, J.-G., Lin, Q., Rajmohan, S., Zhang, D., & Zhang, Q. (2024). Large Action Models : From Inception to Implementation. *arXiv preprint*. https://arxiv.org/html/2412.10047v1

Large Language Models for C Test Case Generation : A Comparative Analysis. (2025). *Preprints*. https://www.preprints.org/manuscript/202505.0399/v1

A.2 Articles de blog et publications techniques

Codewave. (s.d.). Understanding Large Action Models : How They Work. https://codewave.com/insights/understanding-large-action-models/

Sruthy. (s.d.). Top AI Testing Tools for Your AI-Powered Testing. *Software Testing Help*. https://www.softwaretestinghelp.com/top-ai-testing-tools/

Yadav, P. (2025, 5 mars). AI In Software Testing. *TestRigor*. https://testrigor.com/ai-in-software-testing/

Yireh, R. (2025, 21 mars). Selenium Best Practices. *Medium*. https://medium.com/@rabiyireh/selenium-best-practices-4b35105264c1

A.3 Ressources d'entreprises et plateformes

Aegis Softtech. (2024). The Evolution of Software Testing : From Manual to Automated. https://www.aegissofttech.com/insights/the-evolution-of-software-testing/

Applitools. (2025). AI-Powered End-to-End Testing Platform. *Gartner Reviews*. https://www.gartner.com/reviews/market/ai-augmented-software-testing-tools/vendor/applitools/product/applitools-intelligent-testing-platform

Leapwork. (2025). End-To-End Testing : 2025 Guide for E2E Testing. https://www.leapwork.com/blog/end-to-end-testing

Annexe B

Lexique

1. **DOM** - Document Object Model : Représentation structurée d'une page web permettant aux scripts d'accéder et de modifier le contenu, la structure et le style des documents web.
2. **Fragilité des tests** - Tendance des tests automatisés à échouer lors de changements mineurs d'interface, nécessitant des interventions manuelles fréquentes pour maintenir leur fonctionnement.
3. **Recette** - Phase de validation fonctionnelle d'une application où l'on vérifie que le système répond aux exigences spécifiées avant sa mise en production.
4. **LLM** - Large Language Model : Grand modèle de langage basé sur l'intelligence artificielle, capable de comprendre et de générer du texte de manière contextuelle.
5. **IA** - Intelligence Artificielle : Ensemble de technologies permettant aux machines de simuler l'intelligence humaine pour résoudre des problèmes complexes.
6. **IDE** - Integrated Development Environment : Environnement de développement intégré fournissant des outils complets pour la programmation (éditeur, débogueur, compilateur).
7. **LAM** - Large Action Model : Grand modèle d'action capable d'exécuter des séquences d'actions dans des environnements numériques, évolution des LLM vers l'action concrète.
8. **Android** - Système d'exploitation mobile développé par Google, basé sur le noyau Linux et destiné aux appareils mobiles.
9. **Agent exécutif** - Composant logiciel autonome capable d'effectuer des actions automatisées dans un système, fonctionnant de manière indépendante selon des règles prédéfinies.
10. **Spécification fonctionnelle** - Document détaillé décrivant les fonctionnalités attendues d'un système, servant de référence pour le développement et les tests.
11. **Langage naturel** - Langage humain utilisé spontanément pour communiquer (français, anglais, etc.), par opposition aux langages formels ou de programmation.
12. **Sélecteur** - Moyen technique d'identifier précisément un élément dans une page web (CSS, XPath, ID, classe, etc.) pour l'automatisation des tests.
13. **Pydantic** - Bibliothèque Python spécialisée dans la validation et la sérialisation de données, utilisant les annotations de type Python.

14. **E2E** - End-to-End : Tests de bout en bout validant le fonctionnement complet d'une application du point de vue utilisateur final.
15. **CI/CD** - Continuous Integration/Continuous Deployment : Pratiques de développement automatisant l'intégration du code et le déploiement des applications.
16. **DevOps** - Méthodologie combinant développement logiciel (Dev) et opérations informatiques (Ops) pour accélérer la livraison de logiciels de qualité.
17. **API** - Application Programming Interface : Interface de programmation permettant à différents logiciels de communiquer entre eux de manière standardisée.
18. **JSON** - JavaScript Object Notation : Format léger d'échange de données structurées, facilement lisible par les humains et les machines.
19. **HTML** - HyperText Markup Language : Langage de balisage standard pour créer et structurer le contenu des pages web.
20. **CSS** - Cascading Style Sheets : Langage de style décrivant la présentation visuelle des documents HTML (couleurs, polices, mise en page).
21. **JavaScript** - Langage de programmation interprété, principalement utilisé pour rendre les pages web interactives côté client.
22. **Python** - Langage de programmation polyvalent, apprécié pour sa syntaxe claire et sa richesse en bibliothèques, particulièrement en IA.
23. **Framework** - Structure logicielle réutilisable fournissant des fonctionnalités de base pour développer des applications dans un domaine spécifique.
24. **WebDriver** - Interface standardisée (W3C) permettant aux programmes d'automatiser et de contrôler les navigateurs web de manière programmatique.
25. **Selenium** - Suite d'outils open-source pour l'automatisation des tests web, pionnier dans le domaine depuis 2004.
26. **Cypress** - Framework moderne de test end-to-end pour applications web, s'exécutant directement dans le navigateur pour une meilleure expérience développeur.
27. **Playwright** - Outil d'automatisation web développé par Microsoft, utilisant le Chrome DevTools Protocol pour des performances optimales.
28. **TestCafe** - Framework de test web utilisant une architecture proxy, éliminant le besoin de pilotes spécifiques aux navigateurs.
29. **Prompt engineering** - Art et science d'optimiser les requêtes envoyées aux modèles de langage pour obtenir des réponses plus précises et pertinentes.
30. **Structured outputs** - Technique avancée forçant les modèles d'IA à produire des réponses dans un format structuré prédéfini (JSON, XML, etc.).
31. **Self-healing** - Capacité d'un système à détecter automatiquement ses dysfonctionnements et à appliquer des corrections sans intervention humaine.
32. **Multi-agents** - Architecture distribuée utilisant plusieurs agents logiciels spécialisés coopérant pour accomplir des tâches complexes.
33. **Auto-réparation** - Mécanisme automatique de détection et correction des erreurs dans les tests, réduisant la maintenance manuelle.
34. **Computer Vision** - Domaine de l'IA permettant aux machines d'interpréter et d'analyser le contenu visuel (images, vidéos).

35. **NLP** - Natural Language Processing : Branche de l'IA traitant de l'interaction entre ordinateurs et langage humain naturel.
36. **Gherkin** - Langage de spécification lisible utilisant des mots-clés simples (Étant donné, Quand, Alors) pour décrire le comportement logiciel.
37. **BDD** - Behavior-Driven Development : Méthodologie de développement centrée sur le comportement attendu du logiciel, favorisant la collaboration équipe.
38. **TDD** - Test-Driven Development : Pratique de développement où les tests sont écrits avant le code de production, guidant la conception.
39. **POM** - Page Object Model : Pattern de conception structurant les tests en encapsulant les éléments et actions de chaque page dans des classes dédiées.
40. **Factory Pattern** - Pattern de conception créationnel centralisant la création d'objets, permettant de choisir le type d'objet à instancier dynamiquement.
41. **Strategy Pattern** - Pattern de conception comportemental permettant de sélectionner des algorithmes ou stratégies à l'exécution.
42. **Command Pattern** - Pattern de conception encapsulant une demande en tant qu'objet, permettant de paramétrer, mettre en file d'attente ou annuler des opérations.
43. **XPath** - Langage de requête puissant pour naviguer et sélectionner des nœuds dans des documents XML ou HTML via leur structure hiérarchique.
44. **Chrome DevTools Protocol** - Interface de communication bidirectionnelle permettant d'inspecter, déboguer et contrôler le navigateur Chrome programmatiquement.
45. **SaaS** - Software as a Service : Modèle de distribution logicielle où les applications sont hébergées dans le cloud et accessibles via internet.
46. **On-premise** - Solution logicielle installée et hébergée localement dans l'infrastructure de l'organisation, par opposition au cloud.
47. **Provider** - Fournisseur de service ou de technologie, notamment dans le contexte des modèles d'IA (OpenAI, Anthropic, etc.).
48. **Token** - Unité élémentaire de traitement dans les modèles de langage, représentant généralement un mot, une partie de mot ou un caractère.
49. **Température** - Paramètre d'hyperparamétrage contrôlant le niveau de créativité et d'aléatoire dans les réponses des modèles d'IA (0 = déterministe, 1 = créatif).
50. **Timeout** - Délai d'attente maximal défini pour une opération, après lequel celle-ci est considérée comme échouée si elle ne s'est pas terminée.
51. **Memory Manager** - Composant système gérant le stockage et la récupération de variables partagées entre différents agents ou processus.
52. **Output Manager** - Gestionnaire centralisé des sorties système, logs et messages, assurant un affichage cohérent et une traçabilité complète.
53. **HTMLCleaner** - Module spécialisé dans le nettoyage et l'optimisation du contenu HTML pour améliorer son traitement par les modèles d'IA.
54. **Orchestrateur** - Composant central de coordination dans une architecture multi-agents, gérant la distribution des tâches et la synchronisation.
55. **Step Corrector** - Agent spécialisé dans la détection et la correction automatique des étapes de test qui échouent lors de l'exécution.

56. **Sitemap Agent** - Agent autonome chargé d'explorer et de cartographier la structure complète d'un site web pour optimiser les tests.
57. **Validation Agent** - Agent dédié au contrôle et à la validation des résultats de test, vérifiant la conformité aux attentes définies.
58. **Navigation Agent** - Agent spécialisé dans la gestion des transitions et déplacements entre pages ou états d'une application web.
59. **Action Agent** - Agent exécutif responsable de la réalisation concrète des actions utilisateur (clics, saisies, navigation) sur l'interface.
60. **Logger Agent** - Agent de journalisation centralisant l'enregistrement des événements, erreurs et résultats pour le suivi et le diagnostic.
61. **NLP Agent** - Agent spécialisé dans le traitement du langage naturel, transformant les descriptions humaines en instructions techniques exploitables.