

RÉPUBLIQUE DU BÉNIN
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ D'ABOMEY-CALAVI

INSTITUT DE FORMATION ET DE RECHERCHE EN INFORMATIQUE

BP 526 Cotonou Tel : +229 21 14 19 88
http://www.ifri-uac.net Courriel : contact@ifri.uac.bj



MÉMOIRE

pour l'obtention du

Diplôme de Master en Génie Logiciel

Option : Informatique

Présenté par :
Prénom **NOM**

Génération de tests web à partir de cahiers de recettes hétérogènes par agent

Sous la supervision :
Prenom **Nom**

Membres du jury :

Nom et prénoms du président	Grade	Entité	Président
Nom et prénoms de l'examinateur	Grade	Entité	Examinateur
Nom et prénoms du rapporteur	Grade	Entité	Rapporteur

Année Académique : –

Sommaire

Dédicace ii

Remerciements iii

Résumé iv

Abstract v

List of Figures vi

List of Tables vii

Liste des Algorithmes viii

Liste des acronymes ix

Introduction Générale 1

I Généralités sur le thème 6

II Matériels et méthodes 21

III Résultats et discussions 28

Conclusion Générale et Perspectives 32

Références bibliographiques 33

Références webographiques 35

Table des matières 36

Dédicace

Mes dedicaces

Remerciements

Nos remerciements

Résumé

Resume en francais

Mots clés : ....,....,....

Abstract

Resume en anglais

Key words: ....,....,....

List of Figures

List of Tables

I.1 Les outils d'automatisation classique 8
I.2 Matrice de positionnement des travaux 19

II.1 Taxonomie des scénarios de cahier de recettes 22
II.2 Pipeline de l'agent intelligent 23

III.1 Conversion des scénarios par type (valeurs à compléter) 28
III.2 Comparaison des trois bras (valeurs à compléter) 29
III.3 Détection des fautes par famille (valeurs à compléter) 30
III.4 Coûts comparés (valeurs à compléter) 30

Liste des Algorithmes

Liste des acronymes

acroglo :

A la fois dans les deux 1, *Glossaire :* acroglo

acronyme :

Def acronyme 1

Introduction Générale

Contexte et motivation

Le développement logiciel moderne est marqué par des évolutions fréquentes qui nécessitent une validation continue des applications avant leur mise en production. Les **tests de régression**, qui vérifient qu'une modification n'altère pas les fonctionnalités préexistantes [16], constituent le filet de sécurité de ce rythme de livraison. Leurs enjeux sont bien connus : sans eux, chaque changement expose l'application à une régression silencieuse ; avec eux, mais exécutés manuellement, le coût de la campagne de validation devient rédhibitoire à chaque version.

L'automatisation de ces tests est un problème ancien et largement résolu du point de vue de l'exécution. Des outils comme Selenium, Cypress ou Playwright permettent de rejouer un scénario sur une application de manière reproductible et d'en tirer un verdict fiable. Mais cette automatisation repose sur la conception et la maintenance de scripts : elle requiert un profil technique, du temps d'écriture, et un effort de mise à jour proportionnel à la fragilité des sélecteurs lorsque l'interface évolue [6, 11]. Le coût global de cette approche est documenté, et son retour sur investissement n'apparaît qu'après plusieurs versions d'une même application.

Or les comportements attendus d'une application ne sont pas écrits pour la machine. Ils sont décrits, par des profils métier, dans des **cahiers de recettes** : documents en langage naturel, hétérogènes dans leur forme — tableaux regularisés, listes numérotées, prose libre, captures d'écran — et vivants, puisqu'ils évoluent sans discipline de versionnement. La transformation de ces documents en tests exécutables reste aujourd'hui largement manuelle. C'est le goulot d'étranglement du processus : l'automatisation commence après la conversion qu'elle prétend économiser. Ce mémoire appelle le premier de ces deux problèmes la *conversion* et le second l'*exécution* ; il se place délibérément sur le premier.

Les avancées récentes des grands modèles de langage ouvrent une perspective nouvelle : une machine capable de lire un cahier de recettes écrit pour un humain, d'en extraire une représentation structurée, de la traduire en tests exécutables et de remettre des rapports exploitables par une équipe de validation. Ce travail explore cette hypothèse avec une exigence constante : que la transformation soit **mesurable**, faute de quoi elle n'est qu'une démonstration de faisabilité sans valeur de preuve.

L'état de l'art

La littérature sur l'automatisation des tests s'ordonne en trois familles. La première regroupe les outils classiques de test d'interface (Selenium, Cypress, Playwright) : l'exécution y est maîtrisée, mais la conversion du cahier de recettes en script demeure un travail humain dont aucune étude de coût ne rend compte. La deuxième famille concerne la génération de tests à partir de langage naturel : les travaux rapportés ciblent pour l'essentiel des API ou du code unitaire, et évaluent des propriétés de compilabilité ou de recouvrement plus que la conformité à une recette. La troisième famille, la plus récente, applique aux agents de test les grands modèles de langage : ces travaux ancrent la génération dans le DOM de l'application, revendiquent une boucle de rétroaction et une auto-guérison des sélecteurs — mais aucun n'est assorti d'un protocole d'évaluation publié.

La synthèse de ces travaux, présentée au chapitre I, fait apparaître un fait structurel : le quadrant « interface et régression, avec évaluation » est vide. Les systèmes qui ciblent l'interface ne rapportent pas de protocole d'évaluation ; ceux qui évaluent rigoureusement le font sur des API ou du code unitaire. Quatre lacunes en découlent et structurent ce travail : le goulot de traduction n'est ni outillé ni mesuré ; aucun corpus francophone et hétérogène de cahiers de recettes n'a été caractérisé ; la circularité de l'oracle — le fait qu'un test généré par le même modèle que son résultat attendu ne prouve que sa propre cohérence — n'est traitée par aucune architecture identifiée ; et le coût caché de l'auto-guérison, qui peut transformer une régression détectée en succès silencieux, n'est pas mesuré.

Problématique et questions de recherche

Ce mémoire traite de la transformation de cahiers de recettes hétérogènes, rédigés en langage naturel par des profils non techniques, en tests web de régression exécutables et fiables, sans recours scripté préalable — et de la mesure de cette transformation. La question principale est la suivante :

> Dans quelle mesure un agent intelligent peut-il interpréter un cahier de recettes hétérogène rédigé en langage naturel, générer et exécuter des tests web de régression fiables, et remettre des rapports exploitables par une équipe de validation, tout en réduisant le temps et l'effort consacrés à ces tâches ?

Quatre questions secondaires la précisent, chacune adossée à une lacune du chapitre I :

- **Interprétation.** Comment un agent peut-il transformer de manière fiable un cahier de recettes hétérogène — du tableau regularisé à la prose libre — en une représentation structurée dont chaque résultat attendu reste traçable au document d'origine ?
- **Exécution.** Dans quelle mesure l'agent peut-il exécuter automatiquement des parcours utilisateurs et détecter les régressions sans que l'ancre du résultat attendu dépende du générateur lui-même ?
- **Rapport.** Les rapports de tests générés sont-ils compréhensibles et actionnables par une équipe de validation non technique ?
- **Gains.** Quels gains en temps, en effort et en coût d'obtention l'agent apporte-t-il par rapport aux approches à scripts ?

Objectifs

L'objectif général de ce travail est de **concevoir et évaluer un agent intelligent** capable d'interpréter un cahier de recettes rédigé en langage naturel, de générer et d'exécuter les tests de régression web correspondants, et de produire des rapports de tests structurés, sur un cas d'étude contrôlé et reproductible.

Cet objectif se décline en six objectifs spécifiques :

1. analyser les limites des approches actuelles d'automatisation fondées sur des scripts, et situer le présent travail dans l'état de l'art ;
2. caractériser l'hétérogénéité des cahiers de recettes par une taxonomie de scénarios et par la constitution d'un corpus de référence ;
3. concevoir une architecture d'agent dont le cœur est une **représentation intermédiaire** ancrée sur le document d'origine et sur le DOM de l'application ;
4. définir le modèle de transformation du langage naturel vers des actions automatisées, assorti d'un invariant de traçabilité ;
5. développer un prototype exécutant ce pipeline et produisant des rapports exploitables ;
6. évaluer la fiabilité, l'efficacité et l'utilisabilité de la solution, et la comparer aux approches à scripts.

Méthodologie

La démarche est guidée par une exigence : aucune des quatre questions secondaires n'a de réponse mesurée dans l'état de l'art, donc **aucune affirmation de ce mémoire ne doit reposer sur une démonstration ad hoc**. Le dispositif, détaillé au chapitre II, repose sur cinq composantes.

Un corpus de référence. Le projet vise quarante-cinq scénarios de validation, typés selon une taxonomie en six classes — tableau regularisé, prose ordonnée, préconditions implicites, données dynamiques, cas d'exception, références aux diagrammes et captures. Les scénarios proviennent de cahiers de recettes réels et, pour couvrir les types absents des documents réels, de scénarios écrits par le protocole lui-même ; cette distinction est revendiquée explicitement dans la présentation des résultats. Les scénarios normalisés sont rédigés à la main : ils constituent la vérité terrain contre laquelle la sortie de l'agent est mesurée, et ne sont jamais générés par l'outil.

Un cas d'étude contrôlé. La solution est évaluée sur une application open-source auto-hébergée, déployée en local dans un conteneur, rejouable sans réseau et versionné. Ce choix garantit la reproductibilité du protocole, la réinjectabilité de fautes connues et le contrôle complet de l'environnement.

Une interface de modèle abstraite. Le prototype s'appuie sur une couche commune aux fournisseurs de modèle : un modèle principal produit tous les résultats rapportés, un modèle secondaire alimente un unique tableau comparatif. Cette séparation préserve la reproductibilité des chiffres dans le temps.

Trois bras de comparaison. Pour répondre à la question des gains, chaque scénario est exécuté trois fois sur trois bras : un exécutant manuel minuté dans le navigateur, un script écrit à la main, et l'agent proposé. L'ordre d'exécution est randomisé et les séries sont répétées trois fois, car l'exécutant manuel s'améliore entre les passes alors que l'agent non : les deux moyennes et les écarts-types sont rapportés.

Une évaluation par injection de fautes. Vingt fautes connues sont injectées dans six versions du système sous test. Les métriques — couverture des pas du scenario, taux d'hallucination, taux de détection, temps de génération et stabilité — sont stratifiées par type de scénario.

Tout le dispositif obéit à un invariant unique : chaque pas du cahier de recettes est traçable à un pas du test généré, et chaque sélecteur à une observation du DOM. Cet invariant exprime concrètement la parade à la circularité de l'oracle : un test généré dont le résultat attendu n'est adossé à rien ne prouve que la cohérence du modèle avec lui-même, et il est traité comme invalide par construction.

Contributions

- **C1 — Une taxonomie et un corpus.** Une taxonomie en six types de scénarios et la caractérisation d'un corpus francophone et hétérogène de cahiers de recettes, avec sa distribution publiée — un objet absent de l'état de l'art.
- **C2 — Une architecture d'agent ancrée.** Une architecture dont la représentation intermédiaire porte l'invariant de traçabilité, dotée d'une boucle de rétroaction et d'une auto-guérison dont le coût caché est mesuré.
- **C3 — Un protocole d'évaluation contrôlée.** Un dispositif à trois bras adossé à l'injection de fautes et à des métriques stratifiées, applicable au quadrant « interface et régression » laissé vide par les travaux existants.
- **C4 — Le coût caché de l'auto-guérison.** Une mesure des succès silencieux de l'auto-guérison, qui distingue la fiabilité réelle de la fiabilité apparente. *(Contribution d'ambition : le mémoire reste complet sans elle.)*

Plan du mémoire

Le **chapitre I**, consacré aux généralités sur le thème, établit l'état de l'art : notions et vocabulaire, automatisation classique et son coût, génération automatique de tests et question de l'oracle, application des grands modèles de langage, agents auto-guérisseurs, puis la synthèse et les lacunes sur lesquelles ce travail se positionne.

Le **chapitre II** présente les matériels et méthodes. Il caractérise l'hétérogénéité des cahiers de recettes par la taxonomie et le corpus, définit l'architecture de l'agent et sa représentation intermédiaire, décrit les matériels et l'environnement, expose le protocole expérimental à trois bras avec injection de fautes, et formule les menaces à la validité.

Le **chapitre III** présente les résultats et leur discussion : conversion des cahiers en tests, génération et exécution, détection des fautes injectées, coûts et qualité des rapports, puis réponses aux questions de recherche et limites.

La **conclusion générale et les perspectives** reprend les contributions, expose ce qui est établi, ce qui reste ouvert et les prolongements possibles.

Chapitre I

Généralités sur le thème

I.1 Notions et vocabulaire

Ce chapitre utilise un vocabulaire technique précis, dont le développement figurent en introduction générale. Ce paragraphe ne les expose pas : il les définit une fois, pour permettre la lecture autonome des sections qui suivent.

I.1.1 Du cas de test au scénario de validation

Un **cas de test** (ou test case) est un ensemble de conditions décrivant une vérification à mener sur un logiciel, incluant les préconditions, les actions et le résultat attendu [16]. L'**oracle de test** est le mécanisme par lequel le résultat observé est comparé au résultat attendu : c'est lui qui décide si le cas de test passe ou échoue. Un cas de test sans oracle n'est pas exécutable de manière automatique.

Un **scénario de test** est la formulation d'un parcours au sein d'un cas de test. Dans le contexte de la recette fonctionnelle, il correspond à un parcours utilisateur décrit de l'extérieur : une suite d'actions et de résultats observables.

Le **système sous test** (*System Under Test*, SUT) désigne l'application sur laquelle le test est exécuté.

I.1.2 Le cahier de recettes

Un **cahier de recettes** (*recette*, *specification by example*) est le document par lequel un donneur d'ordre — le client, le métier, le responsable produit — formalise les comportements attendus d'une application. Il constitue historiquement le support de la recette : une phase de validation conduite par l'utilisateur final ou son représentant, distincte de la phase de développement.

Trois caractéristiques de cet artefact structurent l'ensemble du présent travail :

Il est rédigé en langage naturel, par un non-développeur. Son auteur est un spécialiste métier, non un ingénieur logiciel. Il n'a pas vocation à être machine-exécutable.

Il est hétérogène dans sa forme. Selon l'organisation, le domaine métier et l'époque, un cahier de recettes prend des formes très diverses : tableaux regularisés, listes numérotées, prose libre, procédures avec captures d'écran, ou encore document Word non structuré. Cette hétérogénéité n'est pas un défaut d'écriture : c'est le reflet de la diversité des métiers qui écrivent.

Il est vivant. Contrairement au code, un cahier de recettes évolue à un rythme propre, sans discipline de versionnement ni revue technique formelle. Un scénario écrit il y a deux ans peut être

obsolète sans que personne ne l'ait remarqué.

I.1.3 Le test de régression

Un **test de régression** est un test exécuté après une modification du système afin de vérifier que les fonctionnalités préexistantes n'ont pas été altérées [16]. Le terme renvoie à la fois au test lui-même et, par extension, à la campagne de revalidation déclenchée par un changement.

La propriété qui rend la régression coûteuse est la suivante : à chaque version livrée, l'ensemble des tests existants doit être rejoué sur une application dont l'interface et le comportement ont pu évoluer. Les tests qui ne résistent pas à cette évolution sont dits **fragiles** (*brittle*) et doivent être réparés, sans quoi ils masquent la régression qu'ils étaient chargés de détecter.

I.1.4 Conversion et exécution : deux problèmes distincts

Ce mémoire distingue deux problèmes que la littérature anglophone regroupe souvent sous l'expression *test automation* :

- la **conversion** — le passage d'une spécification en langage naturel à une forme exécutable par une machine ;
- l'**exécution** — le rejeu de cette forme sur une application, et la production d'un verdict.

Ces deux problèmes sont de nature et de coût dissemblables. L'exécution automatisée est un problème largement résolu, mature et outillé depuis plus de quinze ans. La conversion, en revanche, reste largement manuelle — et c'est elle qui motive ce travail.

I.2 L'automatisation classique des tests : techniques, outils et limites démontrées

I.2.1 Deux familles d'approches

L'automatisation du test fonctionnel web se divise en deux familles historiquement distinctes, que l'on peut comparer sur le plan du coût d'acquisition initiale et du coût de maintenance.

L'approche **capture / rejeu** (*capture and replay*, C&R) consiste à enregistrer les actions du testeur sur l'interface pour générer un script rejouable. Selenium IDE en est l'archétype. Son avantage est une prise en main immédiate : aucune compétence de programmation n'est requise pour produire les tests.

L'approche **programmable** consiste à écrire les cas de test comme des artefacts logiciels, à l'aide d'un langage de programmation et d'une bibliothèque de pilotage du navigateur. Selenium WebDriver, Cypress et Playwright appartiennent à cette famille. Cette approche exige un profil technique, mais produit des tests plus robustes et plus maintenables.

Leotta et al. ont comparé les deux familles sur six applications web réelles [10]. Leur résultat est nuancé et mérite d'être rapporté fidèlement, car il est souvent simplifié dans la littérature de vulgarisation : le test programmable coûte **32 % à 112 % de plus à écrire** que le test par capture, mais sa maintenance coûte **16 % à 51 % moins cher**. Après deux versions environ, l'investissement supplémentaire est rentabilisé. Autrement dit, **le test programmable n'est pas une mauvaise approche : c'est l'investissement initial qui est lourd, et c'est lui qui se paie.**

I.2.2 Les outils

TABLE I.1 : Les outils d'automatisation classique

Outil	Modèle	Positionnement
Selenium IDE	Extension navigateur, capture/rejeu	Prototype rapide, sans compétence
Selenium WebDriver	Bibliothèque, langage libre	Référence historique du test web
Cypress	Exécution dans le navigateur, debug intégré	Bonne expérience développeur, contraintes d'exécution d'exécution
Playwright	Protocole CDP, multi-moteur, auto-attentes	Attentes natives, API programmatique, exécution serveur

Playwright s'est imposé comme le standard de facto de la génération automatisée de tests web par les travaux récents du domaine, pour trois raisons qui sont toutes des propriétés techniques : les **auto-attentes** éliminent une part majeure des tests instables (flaky tests) enévite les temporisations artificielles ; l'**API programmatique** est conçue pour être pilotée par un agent et non seulement par un humain ; l'**exécution sans navigateur visible** convient à un environnement de intégration continue.

Selenium reste majoritaire dans le code industriel existant, ce qui explique que la littérature empirique sur la maintenance des scripts l'ait pris comme objet d'étude [6].

I.2.3 Les patterns de structuration

Page Object Model. Le *pattern Page Object* encapsule les sélecteurs et les interactions d'une page dans une classe dédiée, séparant le quoi du comment. Son bénéfice mesuré sur des suites industrielles atteint environ **70 % de réduction du nombre de modifications de maintenance** lorsque les principes de conception sont correctement appliqués [5].

Tests pilotés par les mots-clés. Le *keyword-driven testing* encapsule les actions derrière des mots-clés réutilisables (`Se connecter`, `Ajouter au panier`) dans une couche d'abstraction séparée. Cette architecture réduit la duplication — mais l'étude de cas de BGL BNP Paribas montre que **plus de 90 % des mots-clés de la suite analysée avaient été modifiés** sur une période de huit mois, et que **plus de 30 % étaient des clones** [5]. Le problème n'est donc pas l'architecture, mais la difficulté à choisir les bons niveaux d'abstraction.

I.2.4 Les langages de spécification : Gherkin et BDD

Le *Behavior-Driven Development* a introduit une spécification textuelle en langage naturel proche du métier, structurée en trois blocs : **Given** (contexte initial), **When** (action), **Then** (résultat attendu) [13]. Cette syntaxe, formalisée dans le langage **Gherkin**, est implémentée par Cucumber, Behave et ses dérivés.

L'apport de cette approche est réel sur le plan de la lisibilité et de l'alignement métier. Il faut cependant en mesurer précisément la portée : le fichier `.feature` **n'est pas exécutable**. Il doit être relié à des *step definitions* écrites dans un langage de programmation, qui contiennent le code réel d'interaction avec le navigateur. Gherkin déplace donc la frontière entre le métier et le développement, **sans supprimer le travail de développement**. La structuration en blocs Given/When/Then constitue en revanche une représentation intermédiaire de grande valeur — et c'est cette représentation que la littérature récente des agents a su réutiliser, comme on le verra en section I.5.

I.2.5 La production de rapports

Les exécutions automatisées produisent des rapports :.junit (format XML brut), **Allure** (rapport lisible riche en métadonnées), ou les rapporteurs HTML intégrés aux frameworks modernes comme Playwright. Ces rapports indiquent l'état de chaque test, la trace d'exécution et les captures d'écran en cas d'échec.

Ces rapports sont conçus pour l'**ingénieur** qui vient de lancer la campagne : ils répondent à la question « qu'est-ce qui a échoué et où ». Ils ne sont pas conçus pour le **diagnostic** : identifier si l'échec provient d'un défaut de l'application ou d'un défaut du test lui-même. C'est un angle mort que le présent travail entend.

I.2.6 Le coût réel de la maintenance des scripts

Les limites de l'automatisation classique ne sont pas une opinion : elles sont quantifiées par des études empiriques de terrain. Les chiffres suivants sont rapportés par les auteurs eux-mêmes.

Obsolescence rapide. Sur un corpus de huit projets open source utilisant intensivement Selenium, **75 % des fichiers de test ont une durée de vie limitée à trois modifications ou moins**, après lesquelles ils sont supprimés ou modifiés au-delà de toute reconnaissance [6]. Autrement dit, la majorité de la valeur d'un test Selenium est consommée en quelques révisions.

Casses généralisées. Sur Acrobat Reader, **74 % des scripts de test d'interface cessent de fonctionner entre deux versions successives** ; une simple modification mineure de l'interface peut rendre défectueux **30 % à 70 % des scripts** d'un système [8]. L'identification et la réparation de ces scripts endommagés incombait alors aux ingénieurs de test, sans outillage adéquat.

Coût financier. Le même travail rapporte un coût de diagnostic de scripts cassés de l'ordre de **120 millions de dollars par an** pour une seule entreprise [8]. Ces chiffres, bien que datés, illustrent l'ordre de grandeur du problème et sont depuis régulièrement repris comme référence.

Sélection des sélecteurs. Le choix du localisateur d'élément détermine directement la fragilité. Sur les suites analysées par Christophe et al., les localisateurs par `id` se sont révélés les plus robustes, tandis que **environ 60 % des localisateurs XPath** avaient dû être réparés [6]. Ce résultatanecdotique est en réalité un enseignement majeur pour la conception d'un agent : **la valeur d'une stratégie de sélélecteur est mesurable et dépendante du contexte applicatif**.

I.2.7 Ce que l'analyse coût-bénéfice dit réellement

Il serait intellectuellement malhonnête de présenter l'automatisation classique comme un échec. Les travaux de Leotta et al. établissent que le test programmable devient rentable après environ deux versions, et que le coût dominant est l'**implémentation initiale**, qui représente près de 87 % du coût total avant d'atteindre le seuil de rentabilité [11]. Cette même étude souligne que Selenium **exige un profil de programmation** là où les outils par reconnaissance visuelle n'en exigent pas.

La lecture correcte est donc la suivante, et c'est celle que ce mémoire adopte : le test scripté n'est pas trop cher **une fois écrit**. Il est coûteux **au moment de l'écrire**, et cette écriture suppose une compétence que le métier n'a pas.

I.2.8 Lacune : le coût de la traduction n'est ni outillé ni mesuré

Toutes les approches décrites ci-dessus partagent une **présupposition commune** : le scénario de test existe déjà sous une forme directement exécutable, ou directement transformable en une telle forme.

Cette présupposition est satisfaite dans deux cas seulement : soit un ingénieur testeur a déjà écrit le scénario en script ou en mots-clés — auquel cas le coût de l'écriture est justement celui que Leotta et al. ont mesuré ; soit le cahier de recettes a été manuellement traduit en fichier Gherkin — auquel cas on a déplacé le travail plutôt que supprimé, et ce travail de traduction n'apparaît dans aucune étude de coût que nous ayons relevée.

Lautomate commence donc toujours après le travail qu'il est censé économiser. Dans une organisation où les scénarios sont formulés en langage naturel par des profils métier, le volume de travail situé en amont de l'automatisation — la traduction — constitue un goulot d'étranglement permanent, et c'est un goulot que personne ne mesure.

C'est cette lacune que ce travail attaque : non pas en concurrençant l'exécution automatisée, mais en supprimant l'étape de traduction manuelle qui la précède.

I.3 La génération automatique de tests et le problème de l'oracle

I.3.1 Trois familles de génération

La génération automatique de tests est un domaine ancien et abondamment documenté. On peut la classer selon la source dont le générateur dérive les tests.

Génération à partir du code. C'est la famille la plus mature. Les techniques de couverture de branches ou de couverture de lignes produisent une suite de tests qui exerce tous les chemins du programme. Le *mutation testing* va plus loin en introduisant des fautes dans le code et en cherchant les tests qui les détectent : un mutant survivant révèle un trou dans la suite. Le *fuzzing* alimente des entrées pour atteindre des chemins d'exécution non couverts.

Ces techniques sont efficaces mais **aveugles**. Elles produisent des tests qui satisfont un critère de couverture, non des tests qui vérifient un besoin. Une suite atteignant 100 % de couverture de branches peut parfaitement laisser passer un défaut fonctionnel majeur, simplement parce que le critère de couverture ignore ce que l'utilisateur attend.

Génération à partir de spécifications formelles. La génération à partir de modèles (Statecharts, réseaux de Petri, modèles de Markov) produit des tests dont la terminaison et la couverture des états peuvent être garanties mathématiquement. Cette approche est puissante et formellement claire, mais elle exige un modèle du système, c'est-à-dire un artefact que l'on doit construire et maintenir — coût qui dépasse souvent le bénéfice obtenu.

Génération à partir d'une spécification en langage naturel. C'est la famille la plus récente, et celle qui nous concerne. Un spécificateur en langage naturel exprime une intention métier directement dans un format lisible par tous, sans modèle formel ni compétence technique.

I.3.2 Le problème de l'oracle

Le problème central de la génération automatique de tests n'est pas la **production** du test : c'est la **vérification** de son résultat.

Produire une séquence d'actions est un problème résolu. Savoir si le résultat obtenu est correct est un problème beaucoup plus difficile, et la littérature consacre un terme dédié : l'**oracle de test**. Le rapport d'une revue systématique de 93 travaux sur les tests métamorphiques et les grands modèles de languepresents cette limitation comme structurelle : la nature générative, probabiliste et ouverte

des LLM **aggrave le problème de l'oracle** et limite l'applicabilité des méthodes traditionnelles de vérification [1].

Les techniques de validation existantes répondent à cette difficulté par l'abstention. Le *mutation testing* renonce à prédire le résultat correct et vérifie à la place que le test détecte une variation. Le test métamorphique renonce à prédire le résultat exact et vérifie une **relation** entre plusieurs exécutions : si la relation est violée, il y a un défaut [1]. Ces techniques sont élégantes et efficaces, mais elles demandent au concepteur de fournir, à la main, soit une relation métamorphique, soit une règle de mutation — c'est-à-dire un travail d'ingénierie qui déplace la charge sans la supprimer.

Pour un cahier de recettes rédigé par un métier, aucune de ces ressources n'existe a priori.

I.3.3 L'aggravation par la génération : la circularité de l'oracle

Le problème prend une forme beaucoup plus sérieuse lorsque le test est produit par un modèle de langue.

Un agent générateur reçoit la spécification en langage naturel et produit deux choses : la séquence d'actions **et** le résultat attendu. Mais le résultat attendu n'est pas observé : il est **inventé par le même modèle**, à partir de la même phrase, avec les mêmes biais.

Il en découle une circularité. Si le modèle propose l'action *cliquer sur « Valider »* et attend *« confirmation affichée »*, et que le test passe, ce qui est démontré est uniquement que **le modèle est cohérent avec lui-même** — et non que l'application fait ce qu'on attend d'elle. Un test généré peut ainsi réussir tout en manquant un défaut fonctionnel.

Une revue systématique récente sur les approches pilotées par les tests dans le cycle de développement assisted par LLM formule le point avec une netteté utile : le fait qu'un test passe **n'établit ni l'équivalence comportementale, ni l'efficacité du retour d'information, ni l'adhésion au processus** ; les gains agrégés peuvent au contraire masquer des résultats différents selon les modèles, les tâches et les dénominateurs [12].

Cette critique est forte et n'a pas, à ce jour, de réponse complète dans la littérature. Elle constitue l'une des motivations principales du protocole d'évaluation adopté au chapitre III, où l'**ancrage du résultat attendu sur le texte source** est traité comme une exigence de conception et non comme un détail d'implémentation.

I.3.4 Lacune : la conversion en langage naturel reste non caractérisée

Les techniques de génération à partir du code et de spécifications formelles sont abondamment évaluées. La génération à partir du langage naturel, elle, est plus récente et son évaluation est encore réduite à un agrégat unique : un taux de correspondance globale entre la spécification et le test produit. Ce chiffre agrégé est indispensable, mais il masque ce qui détermine l'intérêt. Comprendre **quels types** de spécifications échouent, et pourquoi, est le point qui fait avancer le domaine. C'est précisément ce que ce travail tente d'apporter, en introduisant une caractérisation explicite des types de scénarios et des métriques stratifiées.

I.4 Les grands modèles de langage appliqués au test logiciel

Cette section n'est pas un cours d'intelligence artificielle. Elle ne retient que ce qui est nécessaire à la compréhension du chapitre II : ce que les LLM ont changé dans l'automa-

tisation des tests, et ce qu'ils n'ont pas changé.

I.4.1 Ce qui distingue un LLM d'un modèle de langue classique

Un modèle de langue classique produit une représentation vectorielle d'un texte, utile pour la classification, la similarité ou l'extraction d'entités. Un grand modèle de langue (Large Language Model, LLM) est un modèle de langue de très grande taille, fondé sur l'architecture *Transformer*, dont la mise à l'échelle branche l'**apprentissage en contexte** : il peut exécuter une tâche nouvelle sans ré-entraînement, à partir d'instructions et d'exemples fournis dans le prompt [15].

Cette propriété est celle qui change tout pour notre sujet. L'automatisation classique des tests exige un modèle *entraîné* sur une tâche, ce qui suppose un corpus annoté et un cycle d'entraînement. Un LLM, lui, peut traduire du langage naturel en JSON structuré **sans entraînement**, à condition de formuler la demande et de contraindre la sortie.

Cette contrainte de sortie est un élément d'architecture, non un détail : l'obtention d'une sortie conforme à un schéma est obtenue par *structured output* ou, à défaut, par validation et ré-émission. C'est cette contrainte qui rend le pipeline du chapitre II concevable.

I.4.2 Le panorama mesuré par la littérature

L'application des LLM au test logiciel a fait l'objet de revues systémiques nombreuses et convergentes. Wang et al. ont analysé **102 études** [15] et dégagent la préparation des cas de test (génération, oracle) et la réparation de programmes comme tâches dominantes. Hou et al. ont couvert **395 études** [9] et montrent une concentration des travaux sur les phases de développement et de qualité, au détriment de l'analyse des exigences et de la conception.

Une revue systématique de **224 études** portant sur les modèles de fondation en ingénierie logicielle affine ce constat en proposant une taxonomie bidimensionnelle — phase du cycle de vie $\times$ capacité du modèle — et retient **huit familles de tâches** de test et d'assurance qualité : génération de tests unitaires, création d'oracle, localisation de fautes, test de régression, test d'interface, triage de bugs, détection de vulnérabilités et QA humain dans la boucle [3].

Deux observations en découlent pour le présent travail.

Les trois familles qui nous concernent — régression, interface, QA dans la boucle — sont les moins mature. Les deux premières sont concentrées dans des travaux sur les API ou le code, où l'espace de vérification est plus contraint que celui d'une interface web. La troisième, la QA humaine dans la boucle, est par nature difficile à automatiser et rarement évaluée.

Le test d'interface et le test de régression sont traités séparément dans la littérature. Or un test de régression d'interface est précisément le cas d'usage du présent mémoire. Cette conjonction constitue l'espace que la littérature laisse le plus ouvert.

I.4.3 L'émergence du raisonnement agentique

La seconde évolution est l'émergence, depuis 2024-2025, de modèles/dotnet capables de raisonnement multi-étapes et d'appel d'outils. La distinction technique est la suivante : là où un LLM produit une réponse, un agent produit une **séquence d'actions** clans un environnement, en utilisant le retour de l'environnement pour décider de l'étape suivante.

Cette capacité est celle qui permet de fermer la boucle entre génération et exécution : générer un test, l'exécuter, constater l'échec, observer l'état réel de l'application, et corriger. Les travaux pionniers du domaine exploitent explicitement cette propriété en s'appuyant sur les retours de compilation et

d'exécution comme ancrage de la génération, plutôt que sur les seules conjectures du modèle sur la sémantique du projet.

C'est cette propriété d'**exécution ancrée** qui distingue un agent d'un simple générateur, et qui structure l'ensemble du chapitre suivant.

I.4.4 Limites connues

Trois limites sont rapportées de manière convergente et doivent être anticipées.

La **non-détermination** : à température non nulle, deux exécutions d'un même prompt divergent. Cela affecte directement la reproductibilité des résultats et impose un paramétrage explicite.

La **coût de dépendance** : un agent qui appelle un modèle à chaque étape de boucle produit un coût qui croît avec la longueur du parcours et avec le nombre de reprises. C'est une variable d'évaluation à part entière, et non un détail d'implémentation.

Le **biais de corpus** : les jeux de données publics sont massivement anglophones et largement orientés *repository mining* — ils sont dérivés de dépôts de code, non de documents métier. Un modèle évalué sur ces jeux n'est pas nécessairement performant sur un cahier de recettes francophone rédigé par un métier.

I.5 Les agents intelligents pour l'automatisation des tests

I.5.1 Définition d'un agent dans ce contexte

Un **agent** est une entité qui percçoit son environnement, décide d'une action, l'exécute, et observe le résultat pour décider de la suite [14]. Raméné au test logiciel, un agent de test se caractérise par quatre propriétés :

- il **observe** l'application réelle — son DOM, son état visible, ses réponses ;
- il **décide** d'une action à partir d'une intention en langage naturel ;
- il **agit** sur l'application ;
- il **apprend de l'échec** en réinjectant le résultat observé dans sa décision suivante.

C'est la quatrième propriété — la boucle de rétroaction — qui distingue un agent d'un générateur de tests. Un générateur produit un test et s'arrête. Un agent produit un test, l'exécute, et **révise son propre travail**.

I.5.2 La décomposition de la spécification en langage naturel

La première brique de tout agent de test est la **transformation du langage naturel en une représentation intermédiaire structurée**. La littérature converge sur deux formalisations.

Les blocs Given / When / Then. Le travail *Sakura* décompose une description naturelle d'un scénario en blocs BDD, sur le modèle de *behavior-driven development* de North [13], et fait de cette représentation l'entrée explicite du pipeline [7]. L'avantage est la clarté : le comportement attendu devient explicite et révisable avant toute génération de code.

Le schéma structuré métier. Le projet *ScenarioAI* procède à l'inverse et plus finement : un agent d'analyse des exigences produit un JSON comportant l'acteur, l'action, les préconditions, le résultat attendu et les cas limites, à partir duquel sont ensuite dérivés un fichier Gherkin, un Page Object Model, un module de pilotage et les définitions d'étapes [17].

Les deux approches convergent sur le point essentiel : **une représentation intermédiaire explicite, validée avant génération.** Cette RI n'est pas un détail d'implémentation : c'est l'objet que le modèle produit et que les étapes suivantes consomment. Elle est aussi ce qui rend la conversion *auditable* — on peut relire la RI et vérifier qu'elle reproduit la spécification, ce qui est impossible avec un script généré en une passe.

I.5.3 L'ancrage : de la cible naturelle au sélecteur réel

La difficulté spécifique de l'automatisation d'interface n'est pas la logique du test : c'est l'**identification des éléments**. Une spécification dit « cliquer sur le bouton Valider ». Produire un sélecteur correct pour ce bouton exige de connaître la page réelle : son rôle sémantique, son texte, son identifiant, son position dans l'arbre d'accessibilité.

C'est le **taux d'hallucination** des LLM, et c'est la première cause d'échec des tests web générés. Un LLM seul produit des sélecteurs plausibles et fréquemment absents du document.

Trois stratégies d'ancrage ont été développées, qui s'opposent dans leur source de vérité.

L'ancrage par analyse statique du code. Sakura localise chaque bloc de la spécification dans des classes et méthodes concrètes du code applicatif, par analyse statique [7]. Cette stratégie est très solide pour les tests unitaires, où le code est la spécification. Elle est moins transposable à une application tierce dont on ne possède pas le code.

L'ancrage par le trafic observé. NL2Test produit des tests de régression d'API à partir d'une description en langage naturel **et** d'une capture de trafic enregistrée pendant l'exécution du scénario. Les valeurs dynamiques sont liées aux réponses plutôt que codées en dur, et les dépendances sont confirmées par un **algorithme déterministe de cohérence de valeurs** [4]. Le principe est remarquable : le modèle interprète, mais un algorithme confirme. Sur 51 scénarios industriels, cette approche atteint un taux de correspondance exacte de **82,4 %** et un brouillon fonctionnellement exploitable dans **98,0 %** des cas, avec un taux d'adoption du code produit en production de **85,4 %** sur 3 196 tests générés en neuf mois.

L'ancrage par exploration du DOM. Les outils d'agent orientés web ouvrent la page réelle, capturent un instantané des éléments interactifs et choisissent un sélecteur parmi ceux **observés** — jamais parmi ceux imaginés. Cette stratégie est la seule applicable à une application dont on ne possède ni le code ni le trafic, et c'est donc celle qu'exige un cahier de recettes tiers.

Le principe qui se dégage des trois approches est le même, et il est structurant : **le LLM propose, un mécanisme déterministe confirme.** Dans NL2Test, la confirmation est la cohérence de valeur. Dans une approche orientée DOM, la confirmation est l'observation directe dans l'instantané du document.

I.5.4 La boucle de rétroaction

Troisième brique : la **validation par exécution et le routage de l'échec**.

Sakura met en œuvre un système multi-agents hiérarchique — un agent de localisation, un agent de composition, un agent superviseur — où l'agent de composition synthétise le code de test puis le valide itérativement par compilation et exécution. Les erreurs de compilation ou d'exécution —

type incompatible, import manquant, assertion en échec — sont renvoyées à l'agent responsable pour correction ciblée. Le point déterminant est que la génération est **ancrée dans le retour réel du système de build**, et non dans les seules conjectures du modèle sur la sémantique du projet [7].

ScenarioAI met en œuvre une validation en deux phases — vérification statique en processus (analyse syntaxique, contrôle de couverture des étapes, vérification croisée des méthodes), puis exécution réelle d'un *dry run* — avec routage de l'échec vers l'agent correspondant, réessais limités, puis **ouverture** (*fail-open*) si le seuil est atteint : les fichiers sont malgré tout écrits, assortis d'un en-tête d'avertissement pour relecture humaine [17].

Ce mode *fail-open* mérite d'être souligné car il est contre-intuitif. Il traduit une position : **un brouillon incomplet qu'un humain peut corriger vaut mieux qu'un échec sans livrable.** La même logique underlie le choix, retenu au chapitre II, de ne pas générer d'étape dont la cible n'a pas pu être ancrée avec suffisance de confiance.

I.5.5 L'auto-guérison et son coût caché

Une quatrième technique mérite un traitement particulier, car elle est très répandue et rarement évaluée pour ce qu'elle coûte réellement.

L'**auto-guérison** (*self-healing*) consiste, lorsqu'un sélecteur ne trouve plus son élément, à demander à un modèle un sélecteur de remplacement sémantiquement équivalent, puis à reprendre l'exécution. Les rapports publiés annoncent des taux de succès d'auto-guérison de l'ordre de **85 %** dans les contextes d'essai, avec un temps de runs agentiques de quelques secondes à une douzaine de secondes.

Le mécanisme est séduisant, et il répond à un problème réel : une part importante des tests cassés l'est pour cause de modification d'interface, non de régression fonctionnelle.

Mais il introduit un **risque que la littérature ne mesure pas**. Un sélecteur qui échoue peut être réparé de deux façons très différentes :

- il peut être réparé sur **l'élément voulu**, qui a simplement changé de sélecteur — c'est le comportement utile ;
- il peut être réparé sur **un autre élément qui existe et qui est sémantiquement proche**, mais qui n'est pas celui que la spécification désigne.

Dans le second cas, le test **réussit sans rien vérifier**. Une régression réelle — un bouton de paiement désactivé, une validation supprimée, un prix mal calculé — devient un succès silencieux. L'auto-guérison a alors converti une détection en non-détection, et ce coût n'apparaît dans aucun des taux de succès publiés, qui ne rapportent que le côté favorable.

Aucun des travaux examinés ne rapporte de **taux de fausses réparations** — c'est-à-dire la proportion d'auto-guérisons qui ont réparé le test sur la mauvaise cible. Ce silence est en soi un résultat : il indique que la question n'a pas été posée.

Ce mémoire la pose explicitement, et elle constitue la contribution d'ambition identifiée au chapitre III.

I.5.6 La génération des rapports de tests

Dernier pan : la **production du rapport**. C'est le moins traité de la littérature et le plus directement visible par un utilisateur final de l'outil.

La plupart des systèmes d'agent s'arrêtent au rapport d'exécution natif de leur framework — rapporteur HTML de Playwright, XML JUnit — qui indique l'état de chaque test sans qualifier la nature de l'échec. Les systèmes les plus aboutis ajoutent une **couche de diagnostic en langage naturel** : le modèle reçoit la trace d'exécution, l'état de la page avant et après l'action, et produit une explication courte, sous la forme d'une question : *est-ce un défaut de l'application ou un défaut du test ?*

C'est une fonctionnalité simple en apparence et redoutablement utile en réalité, parce qu'elle déplace le statut de l'information : le rapport cesse d'être une liste de verdicts pour devenir une **aide à la décision**. C'est précisément ce que la question de recherche 3 du présent travail énonce.

I.5.7 Lacune : deux angles restent non traités

Au terme de cette revue, deux angles sont apparus insuffisamment couverts.

L'oracle reste circulaire. Aucune des architectures examinées ne traite explicitement le fait que le modèle qui produit l'action produit aussi le résultat attendu. La contribution de ce travail est d'ancrer le résultat attendu sur le texte source du cahier de recettes, et d'en mesurer la couverture séparément de celle des actions.

L'interface et la régression ne sont pas évaluées ensemble. Les travaux denses sont soit sur l'interface sans injection de fautes de régression, soit sur la régression API à partir de trafic. Le présent travail se situe à leur intersection, sur un corpus francophone hétérogène.

I.6 L'évaluation des approches d'automatisation des tests

> Cette section a une fonction double : rendre compte des métriques utilisées par la communauté, et identifier les fragilités méthodologiques de ces évaluations. C'est cette seconde fonction qui détermine le protocole expérimental du chapitre III.

I.6.1 Les métriques classiques

L'évaluation d'une suite de tests automatisés repose historiquement sur un noyau stable.

Le **taux de détection** (*fault detection rate*) mesure la proportion de défauts effectivement découverts par les tests. Il est considéré comme la métrique de référence, mais sa mesure suppose de **connaître à l'avance la liste des défauts**, ce qui n'est possible que dans un environnement contrôlé.

Le **taux de faux positifs** (*false positive rate*) mesure la proportion d'échecs signalés alors que l'application est conforme. Cette métrique est souvent négligée, alors qu'elle détermine directement la confiance que l'équipe accorde à l'outil.

La **couverture de code** (*code coverage*) mesure la proportion du code exécutée par les tests. C'est une métrique de *proxy* et non de qualité : elle mesure l'exercice du code, pas la validité de la vérification.

Le **coût de maintenance** — mesuré en temps de travail ou en nombre de modifications par version — est la métrique la plus pertinente pour l'industri, et celle que les études de la section I.2.6 privilégient.

I.6.2 Les métriques propres aux générateurs

L'évaluation d'un système qui *génère* des tests ajoute des métriques qui n'ont pas d'équivalent dans le test classique.

La **compilabilité** (*compilability*) mesure la proportion de tests générés qui compilent et s'exécutent sans erreur technique, indépendamment de leur correction sémantique. Sakura rapporte ainsi des gains de **50 % à 78 %** de compilabilité par rapport à un outil agentique standard, à modèle identique [7].

Le **recouvrement de couverture** (*coverage overlap*) mesure le recouvrement entre la couverture obtenue par le test généré et celle du test écrit par un développeur, considéré comme vérité terrain. Sakura rapporte un gain de **38 % à 66 %** [7].

La **fidélité structurelle** (*structural fidelity*) mesure à quel point la structure du test généré ressemble à celle du test de référence : nombre de méthodes mobilisées, forme des assertions. Sakura rapporte un gain de **31 % à 72 %** [7].

La **correspondance exacte** (*exact match*) mesure la proportion de scénarios pour lesquels le test produit est fonctionnel sans retouche humaine. NL2Test rapporte **82,4 %** de correspondance exacte et **98,0 %** de brouillon exploitable après retouches mineures [4].

Ces métriques sont utiles mais partagent un défaut commun : **elles sont agrégées en un nombre unique.** Un taux global de correspondance exacte de 82 % ne dit rien de la répartition : il peut masquer 98 % sur un type de scénario et 40 % sur un autre.

I.6.3 Les protocoles d'évaluation

Un protocole d'évaluation répond à quatre questions : **sur quoi**, **par rapport à quoi**, **avec quelle vérité terrain**, **avec quel budget reproductible**.

Le **corpus** doit être représentatif du domaine visé. Les corpus publics de génération de tests à partir du langage naturel sont presque tous dérivés de dépôts de code par *repository mining* : les descriptions y sont synthétisées à partir de tests de développeurs [7]. Cette méthode garantit la vérité terrain, mais elle produit des énoncés qui ne ressemblent pas à ce qu'un métier écrit.

La **vérité terrain** s'obtient de deux façons : soit un test écrit par un développeur sert de référence [7], soit un défaut est délibérément injecté dans l'application et sa position exacte est connue [2]. Cette seconde approche, dite **injection de fautes**, est plus rigoureuse pour la mesure du taux de détection, car elle élimine l'incertitude sur ce qui aurait dû être détecté.

Le **coût** doit être rapporté, et pas seulement le temps d'exécution. Les travaux qui comparent plusieurs modèles rapportent systématiquement le rapport qualité / coût, et montrent qu'un modèle plus petit avec une architecture structurée peut dépasser un modèle plus grand générique [7].

La **reproductibilité** est aujourd'hui un point faible reconnu : les modèles évoluent, donc un résultat obtenu avec un modèle donné n'est pas reproductible a posteriori sans épingler le modèle et sa version.

I.6.4 Les fragilités méthodologiques

Quatre faiblesses affectent les évaluations existantes et doivent être traitées explicitement.

La circularité de l'oracle. Comme établi en section I.3.3, lorsque le générateur produit à la fois l'action et le résultat attendu, un test qui passe démontre la cohérence interne du générateur, non la conformité de l'application. La revue systématique de Liang et al. formule le point mordant : le

succès d'un test **n'établit ni l'équivalence comportementale, ni l'efficacité du retour d'information, ni l'adhésion au processus** [12].

L'absence de faux positifs. Les taux de détection publiés ne sont presque jamais accompagnés d'un taux de faux positifs. Un système qui détecte « tout » obtiendrait un rappel parfait tout en étant inutilisable.

Le biais d'évaluation. Lorsque le générateur et l'évaluateur partagent le même modèle, ou que le test de référence a servi à construire le corpus, l'évaluation est biaisée. Le protocole du chapitre III impose que le modèle ne voit jamais le script de référence avant de générer.

L'agrégation. Le rattachement de plusieurs travaux à une même catégorie masque des résultats contradictoires, précisément parce que les corpus, les indicateurs et les seuils diffèrent d'un papier à l'autre.

I.6.5 Lacune : le corpus francophone hétérogène

Deux constats se combinent.

Premier constat : les corpus publics sont anglophones et issus de *repository mining*. Ils décrivent des énoncés au registre technique, non des cahiers de recipes.

Second constat : l'évaluation d'un agent se fait par un chiffre agrégé unique, qui masque la difficulté différenciée selon les types de scénarios.

Aucun travail identifié ne rapporte des résultats **stratifiés par type de spécification**, sur un **corpus de cahiers de recettes réelles**, **rédigés en français**, et **stratifié selon la difficulté** des scénarios (tableau regularisé, prose libre, préconditions implicites, données dynamiques, cas d'exception).

C'est cette lacune que le protocole du chapitre III entend combler, en introduisant une taxonomie des types de scénarios et des métriques stratifiées.

I.7 Synthèse et positionnement

I.7.1 Matrice de positionnement

Le tableau ci-dessous aligne les travaux de référence sur six axes : la nature de l'entrée, la cible du test, l'ancrage de la génération, la présence d'une boucle de rétroaction, l'auto-guérison et la qualification des rapports.

TABLE I.2 : Matrice de positionnement des travaux

Travail	Entrée	Cible	Ancré	Boucle	Auto-guérison	Rapport en langage naturel	Évalué sur
Selenium, Playwright	script	interface	—	non	non	non	coût de maintenance [6]
Cucumber, Gherkin	fichier `.feature`	interface	—	non	non	non	adoption en projet
Cleverest	message de commit	parseurs, interpréteurs	oui (diff)	oui	non	non	bugs trouvés vs fuzzer [2]
Sakura	langage naturel	tests unitaires Java	oui (analyse statique)	oui	non	non	compilabilité, recouvrement, fidélité
NL2Test	langage naturel + trafic	API	oui (cohérence de valeurs)	oui	non	non	51 scénarios industriels [4]
ScenarioAI, agent-qa	langage naturel	interface	oui (instantané DOM)	oui	oui	oui	**aucune évaluation publiée**
Présent travail	**cahier de recettes**	**interface + régression**	**oui (observation DOM)**	**oui**	**oui, et mesurée**	**oui**	**corpus hétérogène stratifié, injection de fautes**

Deux lectures se dégagent.

Le quadrant « interface + régression, avec évaluation » est vide. Les travaux qui ciblent l'interface ne rapportent pas de protocole d'évaluation ; ceux qui évaluent le font sur des API ou du code unitaire. C'est exactement la cellule que ce travail occupe.

L'auto-guérison n'apparaît que dans des travaux non évalués. Tous les travours qui rapportent des métriques solides n'implémentent pas d'auto-guérison ; ceux qui l'implémentent ne rapportent pas de métriques. Cette rupture est significative : elle suggère que l'auto-guérison et l'évaluation sont difficiles à mener conjointement.

I.7.2 La lacune

Trois constats se combinent en une lacune unique, et un quatrième s'y ajoute comme extension.

G1 — Le goulot de traduction n'est ni outillé ni mesuré. Toutes les approches classiques d'automatisation présupposent un scénario déjà présent sous forme machine. Ce scénario provient soit d'une écriture technique — dont le coût est mesuré [11] — soit d'une traduction manuelle depuis le cahier de recettes, dont **aucune étude de coût ne rend compte**. L'automatisation commence donc toujours après le travail qu'elle prétend économiser.

G2 — Aucun corpus francophone et hétérogène n'a été caractérisé. Les corpus publics sont anglophones et issus de *repository mining* : les énoncés y sont synthétisés depuis des

tests de développeurs, non rédigés par des profils métier. La diversité réelle des cahiers de recettes — du tableau regularisé à la prose libre — n'est ni décrite ni évaluée.

G3 — La circularité de l'oracle n'est pas traitée. Lorsque le modèle produit l'action *et* le résultat attendu, un test qui passe démontre la cohérence interne du modèle, non la conformité de l'application [12]. Aucune architecture identifiée n'ancre le résultat attendu sur une source externe au générateur.

G4 — Le coût caché de l'auto-guérison n'est pas mesuré. Une auto-guérison peut réparer un test sur un élément qui existe mais n'est pas la cible visée, transformant une régression détectée en succès silencieux. Les taux de succès publiés ne rapportent que le versant favorable. *(Contribution d'ambition — traitée si le temps du projet le permet, le mémoire reste valide sans elle.)*

I.7.3 Positionnement du présent travail

Le présent travail ne prétend ni surpasser l'exécution automatisée, ni contester la pertinence des scripts de test. Le test scripté reste le standard, et les études de la section I.2.7 établissent qu'il devient rentable après quelques versions.

Il se situe délibérément **en amont** de cette automatisation et sur son **goulot d'entrée** : la conversion d'un cahier de recettes hétérogène, rédigé en langage naturel par un profil non technique, en scénarios de test exécutables et fiables — sans que cette conversion ne soit aujourd'hui outillée ni mesurée.

Trois réponses sont apportées à cette lacune, correspondant aux trois questions de recherche secondaires portant sur l'interprétation, l'exécution et le rapport ; et une quatrième, plus ambitieuse, sur le coût de l'auto-guérison.

I.7.4 Ce que contient le chapitre suivant

Le chapitre II tirera parti de cette synthèse en trois temps : il caractérisera d'abord plus finement les **types de scénarios** que le corpus de cahiers de recettes fait apparaître, en établissant que leur hétérogénéité — et non leur complexité ponctuelle — est ce qui interdit une simple automatisation par templates ; il tirera ensuite de cette caractérisation à une **architecture** concrète, en justifiant chaque choix de conception par la lacune qu'il adresse ; il formulera enfin les **menaces à la validité** que l'évaluation devra traiter.

Chapitre II

Matériels et méthodes

II.1 Problématique opérationnelle et questions de recherche

Le chapitre I a mis en évidence quatre lacunes qui structurent ce travail : le goulot de traduction entre cahier de recettes et test exécutable n'est ni outillé ni mesuré ; aucun corpus francophone et hétérogène de cahiers de recettes n'a été caractérisé ; la circularité de l'oracle n'est pas traitée ; et le coût caché de l'auto-guérison n'est pas mesuré.

L'objectif du présent chapitre est double. D'une part, il caractérise l'hétérogénéité des cahiers de recettes et propose une taxonomie permettant de la stratifier dans l'évaluation. D'autre part, il définit l'architecture de l'agent intelligent retenue pour y répondre, en justifiant chaque choix par les lacunes identifiées.

Un constat préalable oriente l'ensemble : si tous les scénarios étaient propres et homogènes, un parseur à gabarits suffirait et l'usage d'un modèle de langage ne serait pas justifié. C'est précisément l'hétérogénéité des cahiers de recettes — et non leur complexité ponctuelle — qui impose une approche fondée sur le langage naturel et justifie l'intelligence artificielle.

II.2 Caractérisation des cahiers de recettes et taxonomie

II.2.1 Hétérogénéité des cahiers de recettes

Un cahier de recettes est rédigé par un profil métier, non par un ingénieur logiciel. Il n'a pas vocation à être exécutable par une machine. Cette finalité explique sa grande variété de formes : tableaux regularisés, listes numérotées, prose ordonnée, descriptions de préconditions implicites, cas d'exception, ou encore renvois à des captures ou à des schémas. Cette diversité est le reflet des pratiques métier et constitue la principale difficulté de conversion vers des tests automatisés.

II.2.2 Taxonomie retenue

Pour rendre cette hétérogénéité mesurable, six types de scénarios sont retenus. Ils sont ordonnés par difficulté attendue de spécification, ce qui permet une stratification des résultats au chapitre III.

TABLE II.1 : Taxonomie des scénarios de cahier de recettes

Type	Libellé	Difficulté attendue
T1	Tableau regularisé (une action par ligne, colonnes standard)	Facile
T2	Prose ordonnée (phrases numérotées, peu explicites)	Moyenne
T3	Préconditions implicites (contexte nécessaire non écrit)	Difficile
T4	Données dynamiques (valeurs générées : identifiants, horodatages)	Très difficile
T5	Cas d'exception (messages d'erreur, chemins alternatifs)	Moyenne
T6	Diagrammes et captures (renvoi à une image ou à un parcours)	Très difficile

II.2.3 Distribution visée et paires T1–T3

Le corpus de l'étude vise 45 scénarios au total, répartis comme suit : T1×10, T2×8, T3×8, T4×7, T5×7, T6×5. Parmi ceux-ci, 15 paires appariées T1↔T3 sont prévues afin de permettre une comparaison appariée au chapitre III. Les scénarios proviennent de cahiers de recettes réels lorsqu'ils existent ; pour couvrir les types absents, des scénarios sont écrits par le protocole de recherche. Cette distinction entre *réel* et *protocole* est explicitement conservée dans la présentation des résultats.

II.2.4 Invariant de traçabilité

Pour répondre à la circularité de l'oracle, l'architecture impose un invariant fort : chaque pas du cahier de recettes doit être traçable à un pas du test généré, et chaque sélecteur à une observation du DOM réel. Concrètement, chaque étape porte un champ `source_verbatim` (fragment exact du document justifiant le résultat attendu) et, après ancrage, un `ancrage` avec `selecteur`, `strategie`, `confiance` et `snapshot_ref`. Un test dont l'attente n'est adossée à aucune source externe au générateur est considéré comme invalide par construction.

II.3 Corpus de référence

II.3.1 Objectifs

Le corpus constitue la vérité terrain contre laquelle la sortie de l'agent est mesurée. Il est conçu pour être à la fois représentatif de l'hétérogénéité rencontrée et suffisamment structuré pour autoriser des analyses stratifiées par type.

II.3.2 Règles de constitution

Trois règles guident sa constitution. Premièrement, le dossier `corpus/brut/` est immuable : les documents d'origine y sont copiés tels quels, sans modification ni nettoyage. Deuxièmement, `corpus/normalises` contient un fichier `SC-XXX.md` par scénario, rédigé manuellement selon le gabarit défini dans `corpus/_modele` Ces fichiers ne sont jamais générés par l'outil. Troisièmement, chaque scénario est traçable via `corpus/taxonom` (identifiant, type, fichier source, paire éventuelle, provenance *réel* ou *protocole*).

II.3.3 État actuel

À ce stade, le corpus est vide. Aucun document n'a été déposé dans `corpus/brut/` et aucun scénario n'a été rédigé dans `corpus/normalises/`. Des tests marqués `corpus` vérifient la complétude

(45 scénarios), la cohérence entre taxonomie et fichiers, la répartition par type et l'existence des paires attendues. Ils échouent donc volontairement tant que le corpus n'est pas rempli. Ce caractère « à compléter » est documenté explicitement et ne constitue pas un défaut du code.

II.4 Architecture de l'agent intelligent

II.4.1 Principes de conception

L'architecture répond directement aux lacunes identifiées. Elle repose sur quatre principes : la **traçabilité** (chaque résultat attendu adossé à `source_verbatim` et chaque sélecteur justifié par une observation du DOM) ; le **fail-open** (mieux vaut une étape non résolue qu'un sélecteur non justifié) ; la **reproductibilité** (fournisseur déterministe pour les tests, abstraction du modèle, résultats identiques à configuration égale) ; et la **mesurabilité** (séparation nette entre ce qui est produit et ce qui est mesuré).

II.4.2 Représentation intermédiaire

Le cœur de l'agent est la Représentation Intermédiaire (RI) modélisée avec Pydantic. Elle transforme un scénario en langage naturel en une structure exécutable. Les éléments clés sont : `Specification` (scénario global avec `source_verbatim_global`, `type_scenario`, `etapes`) ; `Etape` (action, cible, `resultat_attendu`, `source_verbatim`, `ancrage` optionnel) ; `Cible` (description naturelle, rôle ARIA, texte, `selecteur`, `confiance`, `snapshot_ref`) ; `Ancrage` (sélecteur observé, stratégie, confiance, `snapshot_ref`, horodatage). Un seuil minimal de confiance est appliqué : un ancrage jugé trop peu sûr est refusé et l'étape est marquée non résolue.

II.4.3 Pipeline

L'agent suit un pipeline structuré en six étapes, chacune produisant un artefact traçable.

TABLE II.2 : Pipeline de l'agent intelligent

#	Étape	Entrée/Sortie	Objectif
1	**Ingestion**	Document brut (PDF/XLSX) → `ScenarioBrut`	Extraction fidèle, sans cor
2	**Décomposition**	`ScenarioBrut` → `Specification` + `RapportFidelite`	Transposition NL→RI, co
3	**Ancrage**	RI partielle + snapshot DOM → RI ancrée	Résolution des cibles en s
4	**Génération**	RI ancrée → script Playwright (Python)	Traduction vers code exéc
5	**Exécution**	Script → trace + captures + verdict	Rejeu sur système sous te
6	**Rapport**	Artefacts → rapport HTML	Restitution lisible par équ

L'ingestion signale ses limites via des avertissements (document scanné, tableau partiel, oracle manquant) afin que la décomposition n'invente pas ce qui manque. La décomposition sépare explicitement la `Specification` (ce qui est produit) du `RapportFidelite` (mesure d'adossement au document) — ce qui évite de « corriger » la sortie avant de la mesurer.

II.4.4 Abstraction du fournisseur de modèle

Un fournisseur de modèle de langage est abstrait via `LLMClient`. Deux implémentations coexistent : un client réel (à configurer) et `FakeLLM`, déterministe et sans réseau. Ce dernier n'est pas un simple bouchon de développement : il garantit la reproductibilité des tests, permet de construire la chaîne sans coût et isole les effets dus au fournisseur des effets dus au code. Par ailleurs, l'identifiant de scénario est imposé par le décomposeur (le modèle ne décide pas de `SC-XXX`), ce qui préserve l'intégrité de la table de résultats.

II.4.5 Auto-guérison et coût caché

L'agent intègre une logique d'auto-guérison des sélecteurs (remplacement par un sélecteur plus robuste lorsque l'ancrage initial échoue) et une boucle de rétroaction pour tenter de corriger les échecs d'exécution. Toutefois, cette capacité peut transformer une régression détectée en succès silencieux si elle ré-ancre sur un élément différent de la cible visée. C'est pourquoi le coût caché de l'auto-guérison fait l'objet d'une mesure explicite dans le protocole d'évaluation (contribution C4) : le mémoire reste complet sans cette mesure, mais elle est prévue pour quantifier les « succès silencieux » et distinguer fiabilité apparente et fiabilité réelle.

II.5 Matériels et environnement

II.5.1 Langages et bibliothèques

Le prototype est développé en Python 3.12. Ce choix unique couvre l'ingestion, la représentation intermédiaire, la décomposition, la génération, l'exécution et la production des rapports, ce qui évite les ponts inter-langage et simplifie le harnais d'évaluation. La validation des structures de données est assurée par Pydantic, qui contraint la sortie du modèle de langage à un schéma strict et rend les erreurs de forme détectables avant toute exécution. Les tests du prototype sont exécutés avec pytest et la qualité statique est contrôlée par ruff.

II.5.2 Exécution des tests générés

L'exécution des tests générés repose sur Playwright, piloté depuis Python. Playwright offre une attente automatique des éléments et une API programmatique adaptée à la génération de scripts. Pour l'évaluation locale, le canal du navigateur Chrome déjà installé sur la machine est utilisé, ce qui évite le téléchargement d'un navigateur supplémentaire ; sur une machine neuve, l'installation se fait par la commande `playwright install chromium`. Les traces et les captures produites lors de l'exécution constituent les artefacts du rapport.

II.5.3 Application sous test

L'application sous test est une application web minimale auto-hébergée, écrite avec Flask. Elle expose un formulaire de connexion et un tableau de bord, et sert de support contrôlé à l'injection de fautes. Elle est fournie avec un fichier de conteneur (Docker) pour garantir la reproductibilité, et peut être lancée directement sans conteneur lorsque l'environnement ne le permet pas. Le choix d'une appli-

cation open-source auto-hébergée correspond à la décision D1 du protocole : il garantit le contrôle de l'environnement, la réinjectabilité des fautes et la rejouabilité du dispositif.

II.5.4 Abstraction du fournisseur de modèle

Le fournisseur de modèle de langage est masqué derrière une interface unique, `LLMClient`, dont il existe plusieurs implémentations. Un client réel permet d'interroger un fournisseur configuré ; un client factice, `FakeLLM`, produit des réponses déterministes sans réseau. Ce dernier n'est pas un simple bouchon : il rend les tests reproductibles, permet de construire la chaîne sans coût et isole les effets dus au fournisseur des effets dus au code.

II.5.5 Outillage du corpus

Deux scripts assistent la constitution du corpus sans jamais produire de scénario à la place de l'auteur. Le premier pré-remplit la table de taxonomie à partir de la distribution visée, en conservant la provenance de chaque scénario et en refusant d'écraser une table existante. Le second génère des squelettes de fichiers normalisés à partir du gabarit et des lignes de la taxonomie, sans écraser les fichiers déjà rédigés. Ces outils réduisent les erreurs de forme et laissent la rédaction du contenu à la charge de l'auteur.

II.6 Protocole expérimental

II.6.1 Trois bras de comparaison

Pour répondre à la question des gains, chaque scénario est traité par trois bras. Le bras manuel est un exécutant humain qui rejoue le scénario dans le navigateur ; il constitue la référence du monde réel. Le bras script est un test écrit à la main par un développeur, représentatif de l'approche classique. Le bras agent est le pipeline proposé. Les trois bras partagent la même application, le même corpus et le même environnement. L'ordre de passage est randomisé et chaque série est répétée trois fois, car l'exécutant humain s'améliore entre les passes alors que l'agent non.

II.6.2 Injection de fautes

La détection est mesurée par injection de fautes connues. Vingt fautes sont réparties en six versions du système sous test, à raison de trois à quatre fautes par version, dont quatre régressions d'interface destinées à éprouver l'auto-guérison. Les fautes couvrent cinq familles : fonctionnelle (calcul erroné), validation (champ obligatoire désactivé), navigation (lien vers une page inexistante), régression (bouton désactivé après refonte) et affichage (message d'erreur supprimé). Pour chaque version, la position exacte de chaque faute est connue : c'est la vérité terrain sans laquelle aucun taux de détection ne serait crédible.

II.6.3 Métriques

Les métriques sont stratifiées par type de scénario. Pour la conversion, on mesure le taux d'exécution (tests qui compilent et tournent), la couverture des pas (pas du cahier retrouvés dans le test), le taux d'hallucination (pas citant un élément absent du DOM) et le taux d'omission (pas du cahier

disparus). Pour l'exécution et la détection, on mesure le rappel (fautes détectées sur fautes injectées), la précision (vraies anomalies sur anomalies rapportées), les faux positifs sur application saine, le taux d'auto-guérison des sélecteurs et la stabilité sur trois exécutions. Pour la qualité des rapports, un questionnaire court auprès de testeurs et le temps de diagnostic sont recueillis. Pour les gains, le temps de conversion, le temps de maintenance après changement d'interface, la couverture et le coût par exécution sont comparés entre les trois bras.

II.6.4 Contrôle des variables

L'application, le corpus et l'environnement sont identiques pour les trois bras. Le réseau est coupé pendant les campagnes et les horodatages sont enregistrés. Le testeur humain est la même personne pour toutes les passes. La température du modèle est fixée à zéro afin de réduire la variabilité, et le modèle ainsi que sa version sont épinglés dans le rapport. Ces contrôles visent à attribuer les écarts observés au bras de comparaison et non à des différences d'environnement.

II.7 Menaces à la validité

La validité du dispositif est examinée selon quatre dimensions. La validité interne est menacée par le biais d'apprentissage du testeur humain, corrigé par la randomisation de l'ordre et la répétition des séries, ainsi que par le risque de sur-apprentissage de l'agent, écarté en interdisant au modèle de voir le script de référence avant de générer. La validité de construit est menacée par des fautes trop faciles, que la variété des familles de défauts vise à éviter, et par la circularité de l'oracle, traitée par l'invariant de traçabilité. La validité externe est limitée par le recours à une application unique ; deux à trois scénarios sur un site public sont prévus pour vérifier que la solution n'est pas surajustée à l'application de test. La validité de conclusion est menacée par la non-reproductibilité des modèles ; l'épinglage du modèle et de sa version, ainsi que l'usage d'un fournisseur déterministe pour les tests, visent à la contenir.

Le corpus écrit par le protocole constitue une menace assumée : il est déclaré comme tel et la provenance de chaque scénario est conservée, afin que les résultats obtenus sur ces scénarios ne soient pas confondus avec ceux obtenus sur des cahiers réels. La sélection du cas d'étude est, quant à elle, encadrée par la grille de notation du protocole, qui privilégie la réinjectabilité, le contrôle de l'environnement et la reproductibilité.

II.8 Synthèse du chapitre

Ce chapitre a caractérisé l'hétérogénéité des cahiers de recettes et proposé une taxonomie en six types, assortie d'une distribution visée à 45 scénarios et de 15 paires T1↔T3. Il a posé l'invariant de traçabilité (`source_verbatim` et ancrage DOM justifié) comme réponse à la circularité de l'oracle, décrit les règles du corpus et son état actuel, puis défini l'architecture de l'agent : représentation intermédiaire, pipeline en six étapes, abstraction du fournisseur de modèle, principes de fail-open, de reproductibilité et de mesurabilité. Il a enfin présenté les matériels, le protocole expérimental à trois bras, l'injection de fautes et les menaces à la validité.

Le chapitre III présente les résultats obtenus par ce dispositif : conversion des cahiers en tests, exécution et détection des fautes, coûts et qualité des rapports, puis leur discussion au regard des questions de recherche.

Chapitre III

Résultats et discussions

III.1 Rappel du dispositif et données analysées

Ce chapitre présente les résultats du dispositif défini au chapitre II et les discute au regard des quatre questions de recherche. Les données analysées proviennent du corpus normalisé, des campagnes d'exécution sur les trois bras et des six versions du système sous test utilisées pour l'injection de fautes.

À ce stade, le corpus et les campagnes ne sont pas encore achevés. Les sections qui suivent en fixent la structure, les tableaux et les indicateurs ; les valeurs seront renseignées après les campagnes, et aucun chiffre n'est avancé tant que la collecte n'est pas terminée. Cette précaution est cohérente avec l'exigence de mesurabilité posée au chapitre II : un résultat n'est publié que s'il provient d'une campagne documentée et rejouable.

III.2 Résultats de la conversion

III.2.1 Taux d'exécution et couverture des pas

La conversion est évaluée par le taux d'exécution, la couverture des pas, le taux d'hallucination et le taux d'omission, stratifiés par type de scénario. Ces indicateurs sont déterminés par comparaison entre la représentation intermédiaire produite et le scénario normalisé de référence.

TABLE III.1 : Conversion des scénarios par type (valeurs à compléter)

Type	Scénarios	Taux d'exécution	Couverture des pas	Taux d'hallucination
T1	–	–	–	–
T2	–	–	–	–
T3	–	–	–	–
T4	–	–	–	–
T5	–	–	–	–
T6	–	–	–	–

III.2.2 Effet de l'hétérogénéité

La comparaison entre types doit permettre de vérifier l'hypothèse posée au chapitre II : la difficulté de conversion croît avec l'implicite du scénario, et non avec sa longueur. Les paires T1↔T3, écrites sur un même parcours, fournissent une comparaison appariée qui neutralise les différences de contenu métier.

III.3 Résultats de génération et d'exécution

III.3.1 Comparaison des trois bras

Les trois bras — manuel, script et agent — sont comparés sur le temps de conversion, le taux d'exécution et la stabilité. Les séries répétées permettent de rapporter une moyenne et un écart-type, et non un unique résultat ponctuel.

TABLE III.2 : Comparaison des trois bras (valeurs à compléter)

Indicateur	Manuel	Script	Agent
Temps de conversion	–	–	–
Taux d'exécution	–	–	–
Stabilité (3 exécutions)	–	–	–
Temps de maintenance	–	–	–

III.3.2 Stabilité et reproductibilité

La stabilité mesure la variabilité des résultats sur trois exécutions de la même suite. Elle distingue les échecs déterministes, qui traduisent un défaut de conversion ou d'ancrage, des échecs intermittents, qui relèvent de la synchronisation avec l'application. Cette distinction est nécessaire pour interpréter les taux de réussite.

III.4 Résultats de l'injection de fautes

III.4.1 Détection et précision

La détection est mesurée par le rappel et la précision, à partir des vingt fautes réparties dans six versions du système sous test. La position exacte de chaque faute étant connue, chaque verdict peut être confronté à la vérité terrain.

TABLE III.3 : Détection des fautes par famille (valeurs à compléter)

Famille de faute	Fautes injectées	Détectées	Faux positifs
Fonctionnelle	–	–	–
Validation	–	–	–
Navigation	–	–	–
Régression	–	–	–
Affichage	–	–	–

III.4.2 Auto-guérison et succès silencieux

Les quatre régressions d'interface servent à distinguer l'auto-guérison légitime de la détection manquée. Un sélecteur réparé sur un élément différent de la cible produit un test qui passe alors que la régression existe : c'est un succès silencieux. Le décompte de ces cas permet d'estimer le coût caché de l'auto-guérison, c'est-à-dire l'écart entre la fiabilité apparente et la fiabilité réelle.

III.5 Coûts et effort

III.5.1 Temps et coût par exécution

Le coût est apprécié à trois niveaux : le temps de conversion du cahier en test, le temps de maintenance après un changement d'interface et le coût d'exécution. Pour l'agent, ce dernier inclut le temps machine et, le cas échéant, la consommation de jetons du fournisseur de modèle.

TABLE III.4 : Coûts comparés (valeurs à compléter)

Coût	Manuel	Script	Agent
Conversion (min/scénario)	–	–	–
Maintenance (min/changement)	–	–	–
Coût par exécution	–	–	–

III.5.2 Qualité des rapports

La qualité des rapports est évaluée par un questionnaire court auprès de testeurs et par le temps nécessaire pour identifier la cause racine à partir du rapport seul. Ces deux mesures complètent les indicateurs techniques : un rapport exact mais inexploitable par l'équipe de validation ne remplit pas l'objectif fixé.

III.6 Discussion

III.6.1 Réponses aux questions de recherche

La discussion reprend les quatre questions secondaires. Sur l'interprétation, les résultats doivent établir si la représentation intermédiaire conserve la trace du document d'origine pour chaque résultat

attendu. Sur l'exécution, ils doivent montrer si la détection des régressions reste fiable lorsque l'ancrage ne dépend pas du générateur. Sur les rapports, ils doivent indiquer si l'équipe de validation comprend et exploite le rapport sans assistance. Sur les gains, ils doivent situer l'agent par rapport aux bras manuel et script.

III.6.2 Contributions

Les résultats sont relus au regard des quatre contributions annoncées : la taxonomie et le corpus (C1), l'architecture ancrée (C2), le protocole d'évaluation contrôlée (C3) et la mesure du coût caché de l'auto-guérison (C4). Pour chaque contribution, la discussion distingue ce qui est démontré, ce qui est seulement outillé et ce qui reste à établir.

III.6.3 Limites

Les limites portent sur la taille du corpus, l'usage d'une application unique et la variabilité des modèles de langage. Elles sont explicitées ici plutôt que reportées à la conclusion, afin que chaque résultat soit lu avec la réserve qui lui correspond.

III.7 Menaces à la validité des résultats

Les menaces propres à l'interprétation des résultats sont distinguées de celles qui concernent la conception du dispositif, traitées au chapitre II. La principale est le biais du corpus écrit par le protocole : il est déclaré et la provenance de chaque scénario est conservée, mais il limite la portée des conclusions aux scénarios concernés. La deuxième est la non-reproductibilité du modèle : l'épinglage du modèle et de sa version, ainsi que l'usage d'un fournisseur déterministe pour les tests, réduisent ce risque sans l'annuler. La troisième est la validité externe : une application unique ne permet pas de généraliser les chiffres à tout système web, ce que les scénarios sur un site public visent à atténuer.

Enfin, le biais d'apprentissage du testeur humain est contrôlé par la randomisation et la répétition, mais ne peut être totalement éliminé. Les écarts entre bras sont donc interprétés comme des ordres de grandeur, et non comme des mesures exactes.

III.8 Synthèse du chapitre

Ce chapitre présente les résultats de la conversion, de la génération et de l'exécution, de l'injection de fautes, des coûts et de la qualité des rapports. Il les discute au regard des quatre questions de recherche et des quatre contributions annoncées, puis explicite les menaces qui pèsent sur leur interprétation.

La conclusion générale reprend ces résultats, rappelle ce qui est établi et ce qui reste ouvert, et ouvre les perspectives de recherche et d'application.

Conclusion Générale et Perspectives

Ce travail est parti d'un constat simple : les cahiers de recettes décrivent les comportements attendus d'une application dans une langue et une forme qui ne sont pas celles de la machine, et leur conversion en tests exécutables reste largement manuelle. L'automatisation classique a résolu l'exécution, non la conversion. C'est ce goulot que le mémoire a choisi de traiter, en posant une exigence constante : que la transformation soit mesurable.

Le chapitre I a établi l'état de l'art et identifié quatre lacunes : le goulot de traduction n'est ni outillé ni mesuré, aucun corpus francophone et hétérogène n'a été caractérisé, la circularité de l'oracle n'est pas traitée, et le coût caché de l'auto-guérison n'est pas mesuré. Le chapitre II a répondu sur le plan conceptuel et méthodologique : une taxonomie en six types de scénarios, un corpus de référence et ses règles de constitution, une représentation intermédiaire ancrée sur le document et sur le DOM, un pipeline en six étapes, un protocole expérimental à trois bras et une injection de fautes contrôlée. Le chapitre III en présente les résultats et la discussion.

Quatre contributions structurent le travail. La première (C1) est une taxonomie et un corpus de cahiers de recettes, avec sa distribution publiée. La deuxième (C2) est une architecture d'agent dont la représentation intermédiaire porte l'invariant de traçabilité, dotée d'une boucle de rétroaction et d'une auto-guérison. La troisième (C3) est un protocole d'évaluation contrôlée à trois bras, adossé à l'injection de fautes et à des métriques stratifiées. La quatrième (C4), d'ambition, est la mesure du coût caché de l'auto-guérison, qui distingue la fiabilité réelle de la fiabilité apparente.

Le travail comporte des limites assumées. Le corpus reste partiellement constitué par le protocole lui-même, ce qui restreint la portée de certaines conclusions aux scénarios concernés. L'évaluation repose sur une application unique, dont le choix est justifié mais qui ne représente pas tous les systèmes web. La variabilité des modèles de langage est réduite par l'épinglage du modèle et l'usage d'un fournisseur déterministe pour les tests, sans être totalement annulée.

Plusieurs perspectives prolongent ce travail. La première est l'extension du corpus à d'autres organisations et à d'autres langues, afin d'éprouver la taxonomie hors de son contexte de construction. La deuxième est l'ouverture à d'autres navigateurs et à des applications plus riches, pour mesurer la généralité de l'ancrage. La troisième est l'étude de l'auto-guérison visuelle, qui constituerait un sujet distinct. La quatrième est l'industrialisation du pipeline, avec une intégration aux chaînes d'intégration continue et une mesure du coût à l'échelle d'une équipe. Le mémoire laisse ainsi ouvert un programme de recherche : faire de la conversion des cahiers de recettes un objet mesurable, et non une simple démonstration de faisabilité.

Références bibliographiques

[1] Authors to be completed. Bidirectional empowerment of metamorphic testing and large language models : A systematic survey. *arXiv preprint arXiv :2605.13898*, 2026.

[2] Authors to be completed. Evaluating llm-based regression test generation. *Proceedings of the ACM on Software Engineering (PACMSE)*, 2026.

[3] Authors to be completed. Foundation models in software engineering : A taxonomy, systematic review, and in-depth analysis of testing support. *Information (MDPI)*, 17(1) :73, 2026.

[4] Authors to be completed. Industrial practice of llm-based test case carving and assertion generation. *Proceedings of the 35th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2026.

[5] BGL BNP Paribas, Quality Assurance Team et al. On the evolution of keyword-driven test suites. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2019.

[6] L. Christophe, R. Stevens, C. De Roover, and W. De Meuter. Prevalence and maintenance of automated functional tests for web applications. *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014.

[7] Google LLC and others. Sakura : An approach for generating complex tests from natural language test descriptions. *arXiv preprint arXiv :2606.00530*, 2026.

[8] M. Grechanik, Q. J. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 1 :408–418, 2009.

[9] X. Hou, Y. Zhao, Y. Liu, et al. Large language models for software engineering : A systematic literature review. *arXiv preprint arXiv :2308.10620*, 2023.

[10] M. Leotta et al. An empirical assessment during test case evolution. In *IEEE International Working Conference on Automated Software Testing (WCRE)*, 2013.

[11] M. Leotta et al. Estimating return on investment for gui test automation frameworks. *arXiv preprint arXiv :1907.03475*, 2019.

[12] Y. Liang, C. Gan, R. Ying, H. Wei, Z. Cui, and S. Ni. Test-driven approaches to software engineering with large language models : A survey of phases, tasks, and agent skills. *arXiv preprint arXiv :2609.12012*, 2026.

[13] G. North. Behavior driven development. In *Advances in Agile Software Development*. Springer, 2006.

[14] S. Russell and P. Norvig. *Artificial Intelligence : A Modern Approach*. Pearson, 3 edition, 2010.

[15] J. Wang, Y. Huang, C. Chen, et al. Software testing with large language models : Survey, landscape, and vision. *arXiv preprint arXiv :2307.07221*, 2023.

Références webographiques

[16] International Software Testing Qualifications Board. Istqb glossary of software testing terms. https://istqb-glossary.org/, 2024. Consulté le 6 novembre 2026.

[17] ScenarioAI contributors. Scenarioai : Ai-powered test suite generator. https://github.com/madhurichinthala17/ScenarioAI, 2026. Projet open source ; consulté le 6 novembre 2026.

Table des matières

Dédicace ii

Remerciements iii

Résumé iv

. iv

Abstract v

. v

List of Figures vi

List of Tables vii

Liste des Algorithmes viii

Liste des acronymes ix

Introduction Générale 1

I Généralités sur le thème 6

I.1 Notions et vocabulaire 6

I.1.1 Du cas de test au scénario de validation 6

I.1.2 Le cahier de recettes 6

I.1.3 Le test de régression 7

I.1.4 Conversion et exécution : deux problèmes distincts 7

I.2 L'automatisation classique des tests : techniques, outils et limites démontrées 7

I.2.1 Deux familles d'approches 7

I.2.2 Les outils 8

I.2.3 Les patterns de structuration 8

I.2.4 Les langages de spécification : Gherkin et BDD 8

I.2.5 La production de rapports 9

I.2.6 Le coût réel de la maintenance des scripts 9

I.2.7 Ce que l'analyse coût-bénéfice dit réellement 9

I.2.8 Lacune : le coût de la traduction n'est ni outillé ni mesuré 9

I.3 La génération automatique de tests et le problème de l'oracle 10

I.3.1 Trois familles de génération 10

I.3.2 Le problème de l'oracle 10

I.3.3 L'aggravation par la génération : la circularité de l'oracle 11

I.3.4 Lacune : la conversion en langage naturel reste non caractérisée . . . 11
I.4 Les grands modèles de langage appliqués au test logiciel . . . 11
I.4.1 Ce qui distingue un LLM d'un modèle de langue classique . . . 12
I.4.2 Le panorama mesuré par la littérature . . . 12
I.4.3 L'émergence du raisonnement agentique . . . 12
I.4.4 Limites connues . . . 13
I.5 Les agents intelligents pour l'automatisation des tests . . . 13
I.5.1 Définition d'un agent dans ce contexte . . . 13
I.5.2 La décomposition de la spécification en langage naturel . . . 13
I.5.3 L'ancrage : de la cible naturelle au sélecteur réel . . . 14
I.5.4 La boucle de rétroaction . . . 14
I.5.5 L'auto-guérison et son coût caché . . . 15
I.5.6 La génération des rapports de tests . . . 15
I.5.7 Lacune : deux angles restent non traités . . . 16
I.6 L'évaluation des approches d'automatisation des tests . . . 16
I.6.1 Les métriques classiques . . . 16
I.6.2 Les métriques propres aux générateurs . . . 17
I.6.3 Les protocoles d'évaluation . . . 17
I.6.4 Les fragilités méthodologiques . . . 17
I.6.5 Lacune : le corpus francophone hétérogène . . . 18
I.7 Synthèse et positionnement . . . 18
I.7.1 Matrice de positionnement . . . 18
I.7.2 La lacune . . . 19
I.7.3 Positionnement du présent travail . . . 20
I.7.4 Ce que contient le chapitre suivant . . . 20

II Matériels et méthodes **21**
II.1 Problématique opérationnelle et questions de recherche . . . 21
II.2 Caractérisation des cahiers de recettes et taxonomie . . . 21
II.2.1 Hétérogénéité des cahiers de recettes . . . 21
II.2.2 Taxonomie retenue . . . 21
II.2.3 Distribution visée et paires T1–T3 . . . 22
II.2.4 Invariant de traçabilité . . . 22
II.3 Corpus de référence . . . 22
II.3.1 Objectifs . . . 22
II.3.2 Règles de constitution . . . 22
II.3.3 État actuel . . . 22
II.4 Architecture de l'agent intelligent . . . 23
II.4.1 Principes de conception . . . 23
II.4.2 Représentation intermédiaire . . . 23
II.4.3 Pipeline . . . 23
II.4.4 Abstraction du fournisseur de modèle . . . 24
II.4.5 Auto-guérison et coût caché . . . 24
II.5 Matériels et environnement . . . 24
II.5.1 Langages et bibliothèques . . . 24

II.5.2 Exécution des tests générés . . . 24
II.5.3 Application sous test . . . 24
II.5.4 Abstraction du fournisseur de modèle . . . 25
II.5.5 Outillage du corpus . . . 25
II.6 Protocole expérimental . . . 25
II.6.1 Trois bras de comparaison . . . 25
II.6.2 Injection de fautes . . . 25
II.6.3 Métriques . . . 25
II.6.4 Contrôle des variables . . . 26
II.7 Menaces à la validité . . . 26
II.8 Synthèse du chapitre . . . 26

III Résultats et discussions **28**
III.1 Rappel du dispositif et données analysées . . . 28
III.2 Résultats de la conversion . . . 28
III.2.1 Taux d'exécution et couverture des pas . . . 28
III.2.2 Effet de l'hétérogénéité . . . 29
III.3 Résultats de génération et d'exécution . . . 29
III.3.1 Comparaison des trois bras . . . 29
III.3.2 Stabilité et reproductibilité . . . 29
III.4 Résultats de l'injection de fautes . . . 29
III.4.1 Détection et précision . . . 29
III.4.2 Auto-guérison et succès silencieux . . . 30
III.5 Coûts et effort . . . 30
III.5.1 Temps et coût par exécution . . . 30
III.5.2 Qualité des rapports . . . 30
III.6 Discussion . . . 30
III.6.1 Réponses aux questions de recherche . . . 30
III.6.2 Contributions . . . 31
III.6.3 Limites . . . 31
III.7 Menaces à la validité des résultats . . . 31
III.8 Synthèse du chapitre . . . 31

Conclusion Générale et Perspectives **32**

Références bibliographiques **33**

Références webographiques **35**

Table des matières **36**