FROM DETECTION TO CORRECTION: AUTOMATED WEB ACCESSIBILITY REPAIR USING PROMPT-ENGINEERED LLMS

João Pedro Vital Brasil Wieland

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Claudia Maria Lima Werner

Rio de Janeiro
Julho de 2026

FROM DETECTION TO CORRECTION: AUTOMATED WEB ACCESSIBILITY REPAIR USING PROMPT-ENGINEERED LLMS

João Pedro Vital Brasil Wieland

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientador: Claudia Maria Lima Werner

Aprovada por: Prof. Claudia Maria Lima Werner
Prof. Claudio Miceli de Farias
Prof. Marcos Kalinowski

RIO DE JANEIRO, RJ – BRASIL
JULHO DE 2026

Vital Brasil Wieland, João Pedro

From Detection to Correction: Automated Web Accessibility Repair Using Prompt-Engineered LLMs/João Pedro Vital Brasil Wieland. – Rio de Janeiro: UFRJ/COPPE, 2026.

XXII, 162 p.: il.; 29, 7cm.

Orientador: Claudia Maria Lima Werner

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2026.

Referências Bibliográficas: p. 122 – 129.

1. Web Accessibility. 2. WCAG. 3. Automated Program Repair. 4. Large Language Models. 5. Prompt Engineering. I. Maria Lima Werner, Claudia. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Here's to the crazy ones. The misfits. The rebels. The troublemakers. The round pegs in the square holes. The ones who see things differently. They're not fond of rules. And they have no respect for the status quo. You can quote them, disagree with them, glorify or vilify them. About the only thing you can't do is ignore them. Because they change things. They push the human race forward. And while some may see them as the crazy ones, we see genius. Because the people who are crazy enough to think they can change the world, are the ones who do.

Steve Jobs — Think Different, Apple, 1997

Agradecimentos

À minha família, que me apoiou em cada etapa desta caminhada. Aos meus pais, Claudia e Eduardo, pelo apoio incondicional de sempre. Ao meu irmão Bernardo, pelos bons momentos de troca. Ao meu irmão Leonardo e sua noiva Monique Santos, por me receberem com generosidade em São Paulo a cada evento e viagem que fiz ao longo do mestrado.

À minha orientadora, **Professora Claudia Maria Lima Werner**, pela orientação rigorosa e pela honra de poder ser seu último aluno de mestrado.

Ao **Professor Guilherme Horta Travassos**, pelos ensinamentos ao longo do curso, pela disponibilização da infraestrutura de testes que possibilitou os experimentos deste trabalho e, sobretudo, pela generosidade de manter seu laboratório sempre de portas abertas.

Ao **Professor Claudio Miceli de Farias**, por ter me conectado à bolsa que financiou esta pesquisa e por me acolher no LabNet, ambiente que me proporcionou trocas enriquecedoras com alunos e pesquisadores.

À **Netcon Americas** e à **Fazenda IP Fitipaldi** pelo fundamental apoio financeiro viabilizado por meio da bolsa MAI/DAI, com apoio do **CNPq**, sem o qual este trabalho não teria sido possível.

Aos colegas que colaboraram comigo no projeto da fazenda (Matheus Gomes, Gustavo Freitas, Lincoln Proença, Keith Fabre e Ian Meirelles) pela dedicação e pelo espírito de equipe. Tenho plena confiança de que serão brilhantes no futuro.

Aos membros do **AGI Club** do iFood, pelas valiosas discussões e contribuições ao longo do desenvolvimento deste projeto, que auxiliaram na construção e no refinamento das ideias aqui apresentadas.

Aos amigos que fiz durante o mestrado e que sempre me apoiaram ao longo dessa jornada, em especial Larissa Galeno, Andre Giron, Rodrigo Feitosa, Diego Castro e Arthur Sasse.

Aos amigos do **NAF**, coordenado pelo querido Beto Pimentel, que sempre foi meu porto seguro. Em 2025, proporcionaram-me a alegria única de participar da Giga Aventura Científica: uma experiência que levarei para sempre.

Aos amigos do **Teatro na Sede da Cia dos Atores**, minha válvula de escape durante os momentos mais intensos do mestrado. Em especial, a João Gofman, Julia

Nepomuceno e Luã Rottweiler, que foram meus diretores no palco e amigos na vida.

A todos da **Fu2re**, que sempre me apoiaram e onde tive muitos momentos felizes.

Por fim, mas não menos importante, a todos os meus amigos, em especial Byatryz Nogueira, Maria Napolitani, Yasmin Romi, Andre João Bonifácio, Larrissa Dias, João Bragança e Sabrina Dias, que me ouviram com paciência e carinho durante todos esses anos e que fazem parte de quem eu sou.

A todos vocês, minha gratidão sincera.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

DE DETECÇÃO À CORREÇÃO: REPARO AUTOMATIZADO DE ACESSIBILIDADE WEB COM LLMS BASEADOS EM ENGENHARIA DE PROMPT

João Pedro Vital Brasil Wieland

Julho/2026

Orientador: Claudia Maria Lima Werner

Programa: Engenharia de Sistemas e Computação

95,9% dos websites mais visitados ainda violam as Diretrizes de Acessibilidade para Conteúdo Web (WCAG), embora a detecção automatizada seja madura: o gargalo está na correção, que ainda exige interpretação e implementação manuais. Esta dissertação apresenta o *a11y-autofix*, um framework de reparo automatizado em quatro fases que transforma a saída de quatro scanners em um prompt estruturado de seis componentes, submete-o a um LLM executado localmente e valida cada patch em quatro camadas antes de aplicá-lo a código React/TypeScript.

A avaliação tem três etapas, e uma correção só é creditada quando um novo escaneamento por navegador confirma o desaparecimento da violação. Uma ablação de prompting, sobre um único modelo, mostra o zero-shot superando decisivamente few-shot e chain-of-thought (IFR de 56% contra 14% e 10%). O experimento canônico compara três modelos open-weight (até 7B, 6 GB de VRAM) sobre 168 violações em 33 arquivos: a IFR fica entre 50% e 54%, os modelos são estatisticamente indistinguíveis, nenhum atinge o alvo de 0,70, e a taxa de regressão de 5–12% é interceptada pela camada de preservação de interface antes da aplicação. Uma ablação arquitetural mostra que desligar qualquer componente isolado não altera significativamente a IFR.

Os resultados estabelecem uma linha de base reproduzível e que preserva a privacidade em hardware comum. Na faixa avaliada, a estratégia de prompting produziu diferenças maiores que a troca de modelo, ainda que medida sobre um único modelo. O teto remanescente está nas violações semânticas, que exigem inferência sobre a intenção do usuário e resistiram à automação em todas as configurações testadas.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

FROM DETECTION TO CORRECTION: AUTOMATED WEB ACCESSIBILITY REPAIR USING PROMPT-ENGINEERED LLMS

João Pedro Vital Brasil Wieland

July/2026

Advisor: Claudia Maria Lima Werner

Department: Systems Engineering and Computer Science

95.9% of the world's most visited websites still violate the Web Content Accessibility Guidelines (WCAG), even though automated detection is mature: the bottleneck is correction, which still demands manual interpretation and implementation. This dissertation presents *a11y-autofix*, a four-phase automated repair framework that turns the output of four scanners into a structured six-component prompt, submits it to a locally executed LLM, and validates every patch through four layers before applying it to React/TypeScript code.

The evaluation has three stages, and a fix is credited only when a browser re-scan confirms the violation is gone. A prompting ablation, on a single model, shows zero-shot decisively outperforming few-shot and chain-of-thought (IFR of 56% against 14% and 10%). The canonical experiment compares three open-weight models (up to 7B, 6 GB VRAM) over 168 violations in 33 files: IFR sits between 50% and 54%, the models are statistically indistinguishable, none reaches the 0.70 target, and the 5–12% regression rate is intercepted by the interface-preservation layer before any patch is committed. An architectural ablation shows that switching off any single component does not significantly change the fix rate.

The results establish a reproducible, privacy-preserving baseline on commodity hardware. Within the range evaluated, the prompting strategy produced larger differences than the change of model, though it was measured on a single model. The residual ceiling lies in semantic violations, which require inference about user intent and resisted automation across every configuration tested.

Contents

List of Figures **xiv**

List of Tables **xvi**

List of Abbreviations and Symbols **xix**

Declaração sobre o Uso de IA Generativa **xxi**

1 Introduction **1**

1.1 Context and Motivation 2
1.2 Problem Statement 3
1.3 Research Questions 4
1.4 The *a11y-autofix* Framework 5
1.5 Contributions 6
1.6 Dissertation Structure 8

2 Background and Fundamental Concepts **10**

2.1 Web Accessibility and the WCAG Standard 10
2.1.1 What Accessibility Means in Practice 10
2.1.2 Accessibility as a Quality Attribute 11
2.1.3 The WCAG Framework 12
2.1.4 ARIA and Semantic HTML 12
2.1.5 Violation Prevalence and the Automated Detection Paradox . 13
2.2 Automated Accessibility Detection Tools 13
2.2.1 Pa11y 14
2.2.2 Axe-Core 14
2.2.3 Playwright with Axe-Core 15
2.2.4 ESLint with the jsx-a11y Plugin 15
2.2.5 Multi-Tool Consensus and Its Significance 16
2.3 The Detection–Correction Gap 17
2.3.1 Structure of Detection Tool Output 17
2.3.2 Why Existing Auto-Fix Tools Are Insufficient 17

2.4 Large Language Models for Code Generation . . . 18
2.4.1 Why a Language Model, and Not Something Else . . . 18
2.4.2 Architecture and Pre-training . . . 19
2.4.3 Local Deployment and Privacy . . . 19
2.4.4 Hallucination as a Structural Property . . . 20
2.4.5 Failure Modes When Applied Naively to Repair . . . 21
2.5 AI Agents for Automated Code Repair . . . 22
2.5.1 From LLMs to Agents . . . 22
2.5.2 Multi-Agent Architectures . . . 22
2.5.3 Iterative Repair and Feedback Loops . . . 23
2.6 Prompt Engineering for Code Repair . . . 24
2.6.1 The Role of Prompt Engineering . . . 24
2.6.2 Core Prompting Strategies . . . 24
2.6.3 Structured Prompts for Code Repair . . . 24
2.6.4 Ablation Studies . . . 25
2.7 React and TypeScript as the Target Platform . . . 26
2.8 Summary . . . 26

3 Rapid Review: Artificial Intelligence Agents for Code Repair and Generation 28
3.1 Motivation and Scope . . . 28
3.2 Review Protocol . . . 29
3.2.1 Research Questions . . . 29
3.2.2 Eligibility Criteria . . . 29
3.2.3 Search Strategy . . . 30
3.3 Selection Results . . . 31
3.4 Synthesis of Evidence . . . 32
3.4.1 Synthesis Procedure . . . 32
3.4.2 RQ1: Agentic Systems for Code Repair and Transformation . 33
3.4.3 RQ2: Multi-Agent Architectures for Software Engineering Tasks 34
3.4.4 RQ3: Prompt Engineering and Context Construction for Repair 35
3.4.5 RQ4: Validation Oracles for Automatically Generated Repairs 36
3.4.6 RQ5: Limitations and Research Gaps . . . 37
3.5 Implications for the *a11y-autofix* Framework . . . 38
3.6 Threats to Validity . . . 39
3.7 Summary and Conclusions . . . 40

4 The *a11y-autofix* Framework 42
4.1 Pipeline Overview . . . 42
4.2 Phase 1: Detection . . . 43

4.3 Phase 2: Prompt Engineering . . . 45
4.4 Phase 3: Adaptive Routing and Repair . . . 46
4.5 Phase 4: Validation . . . 48
4.6 Implementation: Modularity, Reuse, and Local Execution . . . 49
4.7 Worked Example: A Single Violation, End to End . . . 51
4.8 Summary . . . 53

5 Evaluation Design 55
5.1 Research Paradigm . . . 55
5.1.1 Research Questions . . . 56
5.2 Formal Hypotheses . . . 57
5.2.1 H1: Effect of Prompt Template Completeness (RQ1) . . . 57
5.2.2 H2: Effect of Prompting Strategy (RQ2) . . . 58
5.2.3 H3: Practical Utility Threshold (RQ3) . . . 58
5.2.4 H4: Differential Repair Success Across Issue Categories (RQ4) 59
5.2.5 H5: Functional Regression Rate (RQ3, Descriptive) . . . 60
5.3 Benchmark Dataset . . . 60
5.3.1 Availability of the Corpus . . . 63
5.4 The Three Studies . . . 64
5.4.1 Ablation Study: Selecting the Prompting Strategy . . . 64
5.4.2 Canonical Experiment . . . 66
5.4.3 Architectural Ablation: Does Each Component Earn Its Place? 67
5.5 Outcome Measures and Oracle Definition . . . 69
5.6 Statistical Analysis Plan . . . 70
5.7 Threats to Validity . . . 70
5.8 Summary . . . 72

6 Experimental Results 74
6.1 Experimental Setup . . . 74
6.1.1 Models Under Evaluation . . . 74
6.2 Benchmark Corpus . . . 76
6.2.1 Corpus Composition . . . 76
6.3 Experimental Protocol . . . 78
6.4 Ablation Study: Choosing the Prompting Strategy . . . 78
6.5 Primary Results . . . 80
6.5.1 Aggregate Metrics Across Three Repetitions . . . 81
6.5.2 Per-Repetition Raw Data . . . 82
6.5.3 Issue Fix Rate, Graphically . . . 82
6.5.4 Quality–Speed Trade-off . . . 83
6.5.5 Computational Efficiency . . . 83

6.6 Violation-File Success Rate . . . 84
6.7 Agent Selection and Invocation . . . 85
6.8 Failure Mode Analysis . . . 86
6.9 Violation-Category Analysis (RQ4) . . . 87
6.10 Architectural Ablation (RQ1) . . . 89
6.11 Statistical Analysis . . . 93
6.11.1 Confidence Intervals and Model Comparison . . . 93
6.11.2 Hypothesis Evaluation . . . 94
6.12 Threats to Validity . . . 95
6.13 Summary of Findings . . . 96

7 Discussion **99**
7.1 Interpreting the Results . . . 99
7.2 Failure Modes and the Automation Ceiling . . . 103
7.2.1 What Happened to Hallucination . . . 103
7.2.2 What Kind of Ceiling Is This? . . . 105
7.3 Deployment and Positioning . . . 107
7.4 Limitations . . . 108
7.5 Implications . . . 109

8 Conclusion **113**
8.1 Summary of the Work . . . 113
8.2 Summary of Research Questions, Contributions, and Hypotheses . . . 114
8.3 Limitations . . . 117
8.4 Future Work . . . 117
8.5 Closing Statement . . . 120

References **122**

A Violation Taxonomy **131**
A.1 Mapping Tables . . . 134
A.1.1 WCAG Criterion to Issue Type . . . 134
A.1.2 Scanner Rule Identifier to Issue Type . . . 134

B Rapid Review Search Protocol **136**
B.1 Search Query . . . 136
B.2 Eligibility Criteria . . . 137
B.3 Screening Outcomes . . . 137
B.4 Corpus Characteristics . . . 138
B.5 Included Primary Studies . . . 138
B.6 Sources Beyond the Systematic Search . . . 143

B.7 Validity Threats of the Review 144

C Dataset Inclusion and Exclusion Criteria **146**
C.1 Repository Selection Criteria 146
C.2 Stratified Sampling Design 146
C.3 Snapshot and Integrity Protocol 148
C.4 Annotation Interface and Protocol 148
C.5 Dataset Quality Metrics 149

D Validation Pipeline Implementation **151**
D.1 Pipeline Overview 151
D.2 Layer 1: Syntactic Validation 151
D.3 Layer 2: Structural Interface Preservation 152
D.4 Layer 3: Domain Compliance 153
D.5 Layer 4: Code Quality Assessment 154
D.6 Retry and Fallback Logic 155

E Framework Implementation Details **157**
E.1 Technology Stack 157
E.2 Module Structure 157
E.3 LLM Client Abstraction 159
E.4 Experiment Checkpointing 159
E.5 Few-Shot Example Library 160
E.6 Replication Package 160

List of Figures

4.1 The four-phase *a11y-autofix* pipeline for a single source file. A dashed red arc represents the bounded retry loop: when a patch fails validation, structured feedback is passed back to the repair phase for up to $R_{\max} = 3$ attempts before the file is recorded as a repair failure. . . . 43

4.2 Activity view of the same pipeline. Three decisions govern what happens to a file: whether any violation was detected at all, which repair tier the routing score selects, and whether the generated patch clears every validation layer. A rejected patch returns to the repair activity with the failing layer's diagnostic appended to the prompt, up to $R_{\max} = 3$ attempts; only a patch that clears all four layers is applied and only then is a fix credited, by re-scan. The 1,216 violation-free files of the corpus exit at the first decision. 44

4.3 The four-layer validation pipeline. Each layer applies progressively deeper checks; rejection at any layer feeds structured failure diagnostics back to the repair agent for a bounded retry. 50

6.1 Issue Fix Rate (IFR) per model per repetition and mean (± 1 SD on the mean bar), with the H3 target (IFR ≥ 0.70) marked. All three models fall well below the target and overlap within one standard deviation of one another; `codellama-7b` has by far the widest run-to-run spread. 83

6.2 IFR vs. MTTR (mean over three repetitions; vertical bars ± 1 SD on IFR). Quality is statistically tied across the three models, so the vertical axis carries no ranking; the figure exposes the latency spread at roughly equal quality, where `qwen2.5-coder-3b` is fastest and `codellama-7b` slowest (and most variable). 84

6.3 Computational efficiency metrics. `CL-7b` = `codellama-7b`, `Q3b` = `qwen2.5-coder-3b`, `Q7b` = `qwen2.5-coder-7b`; error bars on (a) are ± 1 SD over three repetitions. `qwen2.5-coder-3b` achieves the lowest MTTR by a large margin; `codellama-7b` the lowest token cost per fix. 85

6.4 Architectural ablation: per-file Issue Fix Rate by condition (mean over 66 files, with approximate 95% bootstrap intervals). Most ablated conditions sit within about two points of the full-pipeline baseline (dashed) in either direction; only dropping internal validation moves the bar appreciably, and even that is not significant after Holm–Bonferroni. The small ordering visible here is carried by a handful of files, not a broad effect. 90
6.5 Per-category Issue Fix Rate (pooled over three repetitions) across the ablation conditions. The architecture is irrelevant for the near-mechanical categories (alternative text is flat at 0.73; label and ARIA barely move) and powerless for keyboard (0 throughout). The movement is concentrated in *semantic* structure and contrast: removing the WCAG semantic context lifts semantic repair from 0.16 to 0.65, and the earlier internal-validation run reached 0.91 (read with the deletion caveat discussed below). 91

List of Tables

3.1 Rapid Review screening summary 31

6.1 Canonical experimental configuration. 75
6.2 Models evaluated in the canonical experiment. 75
6.3 Distribution of the 168 confirmed accessibility violations in the benchmark corpus by issue type and primary WCAG criterion (33 files, 36 projects). † WCAG 2.5.8 (Minimum Target Size) was introduced in WCAG 2.2, not WCAG 2.1 AA; "Other" aggregates these violations. They are **included in the IFR denominator** (i.e., counted against the framework's score) because they represent real accessibility issues detected in the codebase; the effective scope of the experiment is WCAG 2.1 AA plus criterion 2.5.8. See Section 6.2 for the scope rationale. 76
6.4 Severity distribution of the 168 confirmed violations. 77
6.5 Ablation study results using `qwen2.5-coder-3b` (mean ± SD over three repetitions). IFR denotes the issue-fix rate and ρ the functional-regression rate. 79
6.6 Aggregate primary metrics (mean ± SD over n=3 repetitions, 1,249 files per repetition, 168 confirmed violations). IFR is credited by the re-scan oracle; complete-repair SR is over the 33 violation-bearing files; ρ is the functional-regression rate (H5 target $< 5\%$). Best value per metric in **bold** (for ρ and MTTR, lower is better). 81
6.7 Per-repetition results for each model. IFR = issue fix rate (fraction of 168, re-scan credited); Complete-rep. SR over the 33 violation-bearing files; ρ = functional-regression rate; MTTR = mean time per file (s). 82
6.8 Complete-repair SR: fraction of the 33 violation-bearing files in which all confirmed violations were resolved simultaneously. 84
6.9 Issue-level repair breakdown (Repetition 3, 168 violations). SWE-Agent: primary repair agent. OpenHands: fallback agent. *Pending*: violations not resolved after all retry attempts; att. counts include retries. 86

6.10 Failure-mode counts in the final repetition. "No valid code" = the model returned no extractable patch; L1/L2/L3 = rejection at the syntactic, functional-interface, and domain-compliance layers; "Timeout" = per-file wall-clock limit exceeded. 86
6.11 Per-type IFR, pooled across three repetitions (proportional partial credit). n is the per-repetition count of that issue type in the corpus. "†" marks categories with $n \leq 7$: **descriptive only**. CL = `codellama-7b`; Q3b = `qwen2.5-coder-3b`; Q7b = `qwen2.5-coder-7b`. 88
6.12 Architectural ablation on `qwen2.5-coder-7b`. IFR is the mean per-file fix rate with its 95% bootstrap interval; the test is Wilcoxon signed-rank against the full pipeline, paired by file (n=66), Holm–Bonferroni corrected. 89
6.13 IFR point estimate (mean across 3 repetitions) with hierarchical 95% bootstrap confidence intervals (k=3 repetitions, n=33 files, B=5,000 resamples). 93
6.14 Assessment of the five research hypotheses. IFR values correspond to cross-repetition means. 94
6.15 Consolidated model comparison (cross-repetition means). Arrows indicate direction of preference: ↑ higher is better, ↓ lower is better. **Bold** indicates best value on each metric. CL = `codellama-7b`; Q3b = `qwen2.5-coder-3b`; Q7b = `qwen2.5-coder-7b`. 97

7.1 Failure mode taxonomy: root causes, structural location, and prompt mitigations. "Residual" means the failure mode persists despite the mitigation being active in the current framework. 104

8.1 Synthesis of research question answers, contributions, and hypothesis outcomes. CL = `codellama-7b`; Q3b = `qwen2.5-coder-3b`; Q7b = `qwen2.5-coder-7b`. 114

A.1 Violation taxonomy (Part 1 of 2): categories `alt-text`, `label`, `aria`, and `keyboard`. 132
A.2 Violation taxonomy (Part 2 of 2): categories `focus`, `semantic`, and `contrast`. 133
A.3 Representative WCAG 2.1 success criteria to issue type mapping (40+ criteria; abbreviated here for readability). 134
A.4 Representative axe-core and Pa11y rule identifiers to issue type mapping (55+ rules; abbreviated). 135

B.1 Rapid Review screening summary (PRISMA-style). 137

B.2 Publication characteristics of the 55 included studies. 138

C.1 Inclusion and exclusion criteria for the benchmark corpus. 147
C.2 Dataset quality metrics (QM1–QM8) with thresholds and measurement procedures. 149

D.1 Four-layer validation pipeline: layers, checks, and failure targets. . . . 152
D.2 Configurable parameters for the validation pipeline. 155

E.1 Framework technology stack. 158

List of Abbreviations and Symbols

Abbreviations

ARIA	Accessible Rich Internet Applications
AST	Abstract Syntax Tree
AT	Assistive Technology (e.g., screen readers such as NVDA, JAWS, VoiceOver)
CI/CD	Continuous Integration / Continuous Delivery
CLI	Command-Line Interface
CoT	Chain-of-Thought (prompting strategy)
CV	Coefficient of Variation
DOM	Document Object Model
DSR	Design Science Research
GPU	Graphics Processing Unit
IFR	Issue Fix Rate (fraction of confirmed violations resolved)
JSX	JavaScript XML (React markup syntax)
LBI	Lei Brasileira de Inclusão da Pessoa com Deficiência (Brazilian Inclusion Law, No. 13.146/2015)
LLM	Large Language Model
MTTR	Mean Time To Repair (per file)
POUR	Perceivable, Operable, Understandable, Robust (WCAG principles)
RQ	Research Question
RR	Rapid Review
SD	Standard Deviation
SR	Success Rate

TE	Token Efficiency
TPF	Tokens Per Fix
TSX	TypeScript XML (typed React markup syntax)
VRAM	Video Random-Access Memory
WCAG	Web Content Accessibility Guidelines

Symbols

ρ	Functional regression rate: Layer-2 rejections divided by repair attempts
δ	Cliff's delta, a non-parametric effect-size measure
κ	Cohen's kappa, inter-annotator agreement
θ	Sampling temperature of the language model
$R_{\max}$	Maximum number of repair attempts per file

Declaração sobre o Uso de Inteligência Artificial Generativa

Declaro que, no desenvolvimento desta dissertação, foram utilizadas ferramentas de IA generativa como apoio à codificação (escrita e depuração de código), à escrita (revisão gramatical e de clareza) e à geração de gráficos. Tais ferramentas serviram apenas como instrumento de apoio, sob revisão crítica do autor, que assume total responsabilidade pela concepção, análise, originalidade e correção do conteúdo. Nenhuma ferramenta de IA foi usada para fabricar dados, resultados ou referências.

Chapter 1

Introduction

The web has evolved into the dominant infrastructure through which individuals access essential services such as education, healthcare, employment, and civic participation. An inclusive web requires that the applications and pages composing it be accessible to all users, including the estimated 1.3 billion people worldwide living with some form of disability [1]. Accessibility is therefore a quality attribute in its own right, carrying legal and social obligations that reach well beyond usability or design preference [2]. When accessibility is neglected, the web fails to be a universal platform, systematically excluding a portion of the global population from full digital participation.

Unfortunately, the state of web accessibility remains poor. The WebAIM Million report consistently finds that 95.9% of the top one million websites contain detectable violations of the Web Content Accessibility Guidelines (WCAG) [2], the international standard that operationalises accessible web design. This figure is particularly staggering because automated detection tools for accessibility have matured considerably: Pa11y [3], Axe [4], and Lighthouse [5] can identify hundreds of violation types with high precision, and they are already integrated into mainstream development workflows. (Lighthouse is named here only as a representative of this mature tooling; *a11y-autofix* uses Pa11y, Axe-core, Playwright+Axe, and ESLint-jsx-a11y as its detection layer — see Section 1.4.)

The persistence of violations in the face of reliable detection tools points to a structural problem that this dissertation aims to address directly: there is a gap between detection and correction. Automated tools are excellent at diagnosing what is wrong; they produce no mechanism for fixing it. Every flagged violation ultimately demands a developer's time to interpret the report, research the applicable standard, implement a fix, and validate that the change neither introduces regressions nor new violations. At scale, this manual correction burden is the proximate reason why accessibility problems accumulate faster than they are fixed.

The Brazilian context lends this particular urgency. The 2022 IBGE Census

counts 14.4 million people with disabilities (7.3% of the population), a figure that rises to 18.6 million under broader functional criteria [6, 7]. Article 63 of the *Lei Brasileira de Inclusão da Pessoa com Deficiência* (LBI, Law No. 13.146/2015), also known as the *Estatuto da Pessoa com Deficiência*, has, since 2016, required every website of a company operating in Brazil or of a government body to be accessible [8]; yet fewer than 3% of Brazilian sites comply with accessibility standards [9], and enforcement has stayed weak: penalties exist on paper, but systematic auditing is rare. The 2025 technical standard ABNT NBR 17225 now operationalises the LBI with concrete criteria, signalling stricter accountability ahead [10]. For organisations, automating accessibility correction is therefore shifting from a convenience to an engineering necessity: legal exposure is mounting while the manual correction workload stays intractable at scale.

1.1 Context and Motivation

To understand why the correction step is difficult to automate, it helps to examine what a detection tool actually produces. A typical Pa11y report for an accessibility violation looks like this:

```
{
  "code": "WCAG2AA.Principle2.Guideline2_4.2_4_4.H77",
  "message": "Check link text identifies purpose",
  "selector": "a:nth-child(3)"
}
```

This report tells a developer that a link element fails WCAG criterion 2.4.4 (link purpose). It says nothing about what the current link text is, what the destination URL conveys, what React component encapsulates the element, what framework conventions the project follows, or what a compliant fix would look like in the surrounding code context. A developer can bridge these gaps using expertise and judgment.

Simultaneously, recent advances in Large Language Models specialised for code (Codex [11], CodeLlama [12], Qwen2.5-Coder [13], DeepSeek-Coder [14]) offer a compelling path forward. These models can generate syntactically correct and semantically coherent code from natural language descriptions, and they carry substantial pre-trained knowledge about WCAG, ARIA, and common React patterns. The intuitive hypothesis is that one could simply pass a detection report to an LLM and receive a patch in return.

Qualitative observations collected during framework development showed that this naive approach fails reliably. Without structured context, LLMs generate

plausible-looking but incorrect repairs across several recurring failure patterns: they invent non-existent ARIA attributes (`aria-role="navigation-button"`); they rewrite event handlers unrelated to the violation, introducing functional regressions; they add `alt="decorative image"` where `alt=""` is semantically required; and they apply imperative DOM patterns (`document.getElementById`) inside declarative React codebases. These observations, described in detail in Section 4.3, motivated the six-component template design and informed the a-priori ablation study (RQ1). The underlying issue is that the raw detection output fails to activate and constrain the model's knowledge correctly.

This observation defines the central technical problem of the dissertation: how to systematically transform a structured violation report into a prompt that reliably guides an LLM toward a correct and framework-appropriate patch.

1.2 Problem Statement

The gap between detection and correction in web accessibility stems from three main factors. First, detection tools generate outputs designed for human interpretation rather than automated repair: violation codes, CSS selectors, and short descriptions help identify issues, but lack the contextual information needed to fix them. Second, WCAG is a nuanced standard: its 78 success criteria at level AA cover a broad spectrum of requirements, ranging from straightforward rules (e.g., images must include `alt` attributes) to more interpretive ones (e.g., link text should be meaningful out of context). Third, modern web applications are built using diverse frameworks, each with its own conventions; effective repairs must comply with both WCAG and framework-specific practices, which is difficult without explicit guidance.

One plausible diagnosis is that an LLM working from a bare violation report already has the domain knowledge and lacks *context structure*: the information, format, and specificity that would turn implicit knowledge into a correctly constrained repair. This dissertation adopts that diagnosis as a *working conjecture* and not as a premise. If it holds, the problem is tractable through *prompt engineering*—a systematic, repeatable procedure for enriching violation reports with the right information, in the right format, at the right level of specificity—and the framework described in Section 1.4 is its concrete instantiation.

The conjecture is tested rather than assumed, and it survives only in part. The prompting strategy does move repair outcomes substantially (Chapter 6). But the component-level ablation of the prompt template finds no single component that significantly raises the fix rate, and a residual class of violations resists repair under every configuration examined. Chapter 7 takes up what that combination implies about the limits of context structure as an explanation.

Solving this problem in isolation is insufficient: a prompt engineering framework for code repair must also address *validation*. LLM outputs are non-deterministic and subject to multiple failure modes. A repair pipeline without automated quality gates would propagate hallucinated APIs, structural regressions, and over-engineered solutions into production codebases, an outcome that would be worse than the original violations. The framework must therefore combine prompt construction with a multi-layer validation mechanism that rejects invalid outputs and triggers corrective retries before any patch reaches the codebase.

1.3 Research Questions

This dissertation addresses the detection–correction gap through four research questions that together cover the design, optimisation, comparison, and characterisation of the proposed framework. They are stated below in the same form in which they are operationalised in Chapter 5; the paragraphs that follow give the rationale for each.

RQ1 Does a structured six-component prompt template yield significantly higher repair success rates than ablated versions of itself?

RQ2 Do prompting strategies (zero-shot, few-shot, and chain-of-thought) produce measurable differences in repair success rate and token efficiency?

RQ3 Do distinct locally deployable LLM architectures differ significantly in repair success rate, mean time to repair, and token cost?

RQ4 Is repair success rate associated with violation complexity category, and if so, which categories are most and least amenable to automated repair?

RQ1 addresses how detection tool outputs can be systematically transformed into effective LLM prompts for automated accessibility repair. Effective prompts likely require multiple components operating synergistically; this question covers both the design of a structured prompt template and the measurement of each component's marginal contribution through a controlled ablation, the latter identifying which elements are necessary, which are sufficient, and which interact.

RQ2 compares the prompting strategies (zero-shot, few-shot, and chain-of-thought) on repair success rate and token efficiency. Each strategy offers a different balance between accuracy, token consumption, and generation latency; this question provides empirical guidance on strategy selection as a function of violation type and operational constraints.

RQ3 compares locally deployable LLM architectures on repair quality, speed, and cost. Privacy-preserving local deployment constrains the model space to those runnable on commodity hardware; this question characterises the trade-offs among architectures within that constraint, providing practitioners with quantitative guidance for model selection.

RQ4 identifies which violation categories are most receptive to automated repair and what failure patterns emerge across them. Not all WCAG violations are equally amenable to automated correction; this question identifies the structural determinants of repair difficulty and catalogues the failure modes that emerge, informing both prompt design and realistic expectations for automation.

These four questions are answered to deliberately different depths, and the dissertation is explicit about why, because the evidentiary status of each result is itself part of the contribution. RQ2 is answered confirmatorily, through a controlled ablation that finds a large, statistically significant advantage for one prompting strategy; this is the most firmly established finding, with the caveat that the ablation was run on a single model. RQ3 and RQ4 are answered descriptively: the canonical corpus of 168 confirmed violations supports a clear account of model cost and of the boundary between structurally and semantically resolvable repairs, but it is underpowered for inferential claims about small differences between models or about thinly populated violation categories. RQ1 is answered by the architectural ablation, though with a twist: the template is designed, justified, and demonstrated end-to-end, and the component-level ablation that quantifies each part's marginal contribution finds no single component significantly moving the fix rate, a null that is itself informative once read with the limits of its statistical power. This unevenness is not incidental: the binding constraint on the study is not engineering effort but the scarcity of confirmed real-world violations and the cost of local inference under a 6 GB budget. The chapters that follow are organised to make each question's evidence, and its limits, explicit rather than to present a uniform but misleading confidence across all four.

1.4 The *a11y-autofix* Framework

The research vehicle for addressing these questions is *a11y-autofix*, a four-phase automated repair framework for accessibility remediation in React codebases. The framework translates the four research questions into concrete engineering decisions across five integrated components.

Rather than trust a single scanner, the framework runs four detection tools concurrently (Pa11y, Axe-core, and Playwright+Axe on the rendered DOM, plus ESLint's `jsx-a11y` plugin on the JSX source), then deduplicates their findings, scores

each one by cross-tool agreement, and maps it to its governing WCAG 2.1 criterion. Each confirmed violation is turned into a context-rich, constraint-bounded instruction by the framework's central artefact: a six-component prompt template (role definition, violation context, code context, constraints, retrieved few-shot exemplars, and output-format specification), with the exemplars selected by semantic similarity to the violation at hand.

From there, a two-tier router sends most attribute-level repairs to a primary SWE-agent-style engine and escalates the harder, multi-file cases to an OpenHands-style broad-context engine (in the evaluation, both operate via `DirectLLMAgent`, a custom prompt-based adapter; the SWE-agent and OpenHands CLIs were not installed in the execution environment — see Section 4.4 for the full disclosure). Every generated patch must clear four sequential gates (syntactic validity, structural interface preservation, domain compliance, and code quality), and a rejection at any gate returns to the agent as structured feedback for a bounded retry. An asynchronous, resumable experiment runner evaluates every model–strategy–violation combination in a within-subjects design, the infrastructure behind the comparisons in RQ2 and RQ3. Chapter 4 specifies each of these components in full.

1.5 Contributions

This dissertation makes four primary contributions to software engineering and AI-assisted development. To orient the reader, it is important to distinguish between the components of *a11y-autofix* that are original contributions of this dissertation and those that integrate pre-existing open-source tools. SWE-agent [15] and OpenHands [16] are pre-existing autonomous coding agent frameworks whose *prompt formats and routing roles* inspired the framework's repair engines; in practice, the evaluation used a custom `DirectLLMAgent` adapter rather than their CLIs (Section 4.4). Ollama is a pre-existing local model serving runtime. The original contributions of this dissertation are the orchestration layer: the six-component prompt template, the adaptive routing mechanism, the four-layer validation pipeline, the benchmark corpus, and the experimental protocol. None of these components existed in the pre-existing tools; they were designed and implemented as part of this research.

The first is a structured prompt engineering framework for automated code repair. The framework comprises the six-component prompt template and the four-layer validation pipeline described above. It is designed with domain-agnostic principles: while instantiated and evaluated for web accessibility repair, the template structure and validation architecture are applicable to any quality assurance domain that produces structured violation reports with objectively verifiable correction criteria. Adaptation to new domains requires configuring the violation context and

domain compliance layers, not redesigning the framework.

The second is a three-stage evaluation of locally deployable LLMs on real-world violations. A first study isolates the prompting strategy (RQ2) on a single model (`qwen2.5-coder-3b`) and establishes that zero-shot prompting decisively outperforms few-shot and chain-of-thought for this constrained-repair task on that model; this result selects the strategy carried into the main comparison. Whether the advantage of zero-shot generalises to larger models or other architectures is left open by this study (Section 8.4). The canonical experiment then compares three open-weight code models (Qwen2.5-Coder-3B, Qwen2.5-Coder-7B, CodeLlama-7B) under a 6 GB VRAM constraint over 168 violations confirmed by browser re-scan across 33 files, in three stability-check repetitions per model, addressing RQ3 (model architecture comparison) and RQ4 (violation-category receptiveness) across the Issue Fix Rate and the secondary cost, stability, and regression metrics. A third study, the architectural ablation, switches off the framework's structural components one at a time to answer RQ1, and finds, under file-level paired testing, that no single component significantly changes the fix rate, relocating the performance lever onto the prompting strategy rather than the accumulated scaffolding.

The third is a taxonomy of LLM failure modes in accessibility repair. Systematic analysis of unsuccessful repairs yields a five-category taxonomy: invalid patch synthesis, superficial addressing, API hallucination, over-engineering, and context blindness. Each failure mode is mapped to its structural root cause, the prompt template component that mitigates it, and the validation layer that catches residual instances. The taxonomy identifies the two modes that define the current automation ceiling (invalid patch synthesis and superficial addressing) and distinguishes them from the three modes effectively suppressed by the framework's design.

The fourth is a benchmark dataset and open-source implementation. The benchmark comprises accessibility violations confirmed through a multi-tool consensus detection protocol on stratified open-source React repositories, with pinned commit snapshots and content-addressed integrity. The open-source implementation integrates Pa11y, Axe-core, Playwright, and ESLint-jsx-a11y detection with React/JSX AST processing and the Ollama local model backend. A replication package (including prompt templates, validation implementations, and analysis scripts) is released to facilitate reproduction, benchmarking of future models, and adaptation to other repair domains.

The Principal Finding

Stated once, before the chapters that establish it: roughly half of the accessibility violations that a scanner can detect in real React code can be repaired automatically,

on a consumer GPU, by a 3B open-weight model, with no fine-tuning and no curated training data. Across three models and three repetitions the Issue Fix Rate sits between 50% and 54%, and the three models are statistically indistinguishable from one another.

Three qualifications travel with that number, and each is a finding in its own right. The first is where the leverage lies: how the model is prompted moved the fix rate substantially, while the choice of model within the 3B–7B band did not, and switching off any single structural component of the framework did not either. The second is where the automation stops: the unrepaired half concentrates in violations whose correct repair depends on information the source file does not contain, and no configuration examined here recovered them. The third is what the architecture demonstrably bought: it did not raise the fix rate, but it prevented every regressive patch the models proposed from reaching the codebase, which is a contribution to safety rather than to capability.

The reader who wants only the result can stop here; the chapters that follow establish each of these claims and, as importantly, the limits within which they hold.

Parts of this work have been published. An earlier version of the repair concept and its human-centred design rationale appeared as a workshop paper at the 2nd International Workshop on Human Factors for Software Dependability, co-located with the 36th IEEE International Symposium on Software Reliability Engineering (ISSRE 2025) [17]; it precedes the evaluation reported here and makes no empirical claim. The full framework and empirical evaluation presented in this dissertation have been accepted (early decision) as a full research paper in the Research Track of the 40th Brazilian Symposium on Software Engineering (SBES 2026) [18]. The measurement methodology behind that evaluation, including the unit-of-analysis correction that governs how the architectural ablation is read here, has been accepted at the 1st Workshop on Software Engineering for Agentic Systems (SE4AS), co-located with CBSoft 2026 [19].

1.6 Dissertation Structure

The remainder of this dissertation is organised as follows.

Chapter 2 establishes the foundational concepts: web accessibility and the WCAG standard, the detection–correction gap as a software quality problem, the case for LLM-based agentic repair, and the prompt engineering principles that make the approach viable.

Chapter 3 presents a Rapid Review of 55 studies (2020–early 2026) on AI-agent-based code repair and generation. The review synthesises architectural patterns,

prompt engineering strategies, validation mechanisms, and research gaps, providing the empirical justification for the framework's design choices.

Chapter 4 presents the proposal itself: the *a11y-autofix* framework in detail, phase by phase, from multi-tool detection through prompt assembly and adaptive routing to the four-layer validation gate, closing with a worked example that traces one real violation end to end.

Chapter 5 sets out how that framework was evaluated: the Design Science Research paradigm, the four research questions, the five hypotheses stated a priori, the benchmark dataset construction protocol, the three studies, the outcome measures and oracle, the statistical analysis plan, and the threats to validity.

Chapter 6 presents the experimental results, reporting on each research question with the corresponding statistical tests and effect sizes.

Chapter 7 interprets the results in the context of the research questions, discusses the failure mode taxonomy and its implications for prompt design, and analyses the practical trade-offs among models and strategies from a practitioner perspective.

Chapter 8 summarises the contributions, states the limitations of the work, and proposes directions for future research, including extension to other quality domains, integration with CI/CD pipelines, and the role of human oversight in production repair systems.

Appendices provide supporting materials: the complete violation taxonomy (Appendix A), the Rapid Review search protocol (Appendix B), dataset inclusion/exclusion criteria (Appendix C), and implementation details for the validation pipeline (Appendix D). Material that the reader needs in order to judge the evaluation—the hypotheses, the analysis plan, the oracle definition—is in the body rather than in an appendix; what remains in the appendices is reference material and implementation detail.

Chapter 2

Background and Fundamental Concepts

Closing the detection–correction gap requires understanding four things. First, what the gap actually is: the structure of the WCAG standard (Section 2.1), the mechanics of detection tools (Section 2.2), and the precise reason their output is insufficient for automated repair (Section 2.3). Second, what can bridge it: LLMs carry pretrained knowledge of accessibility standards and React idioms, but only under the right conditions (Section 2.4). Third, what those conditions are: agent architectures that enable multi-step, tool-using repair loops (Section 2.5). And fourth, how to create those conditions systematically: structured prompt engineering (Section 2.6). Each section in this chapter introduces exactly one piece of this argument. By the end of Section 2.6, all the components of the *a11y-autofix* framework will have been motivated.

2.1 Web Accessibility and the WCAG Standard

2.1.1 What Accessibility Means in Practice

Because the rest of this dissertation treats accessibility violations as objects to be detected and repaired in source code, it is worth establishing first what those violations mean for the people affected by them. A reader who has never used a screen reader can otherwise be forgiven for reading `alt=""` as a formatting detail rather than as the difference between an image that is silently skipped and one that interrupts a user with a filename.

Consider how a blind user reaches a product page. A screen reader—NVDA, JAWS, or VoiceOver—does not read the screen; it reads the *accessibility tree*, a parallel structure the browser derives from the DOM in which every element carries a name, a role, and a state. The user moves through this tree with the keyboard,

often jumping between headings, landmarks, or form controls rather than reading linearly. Everything that user perceives is therefore mediated by properties that live in the markup: an `<img>` without `alt` has no name, so the screen reader announces the file path or nothing at all; a `<div>` wired with a click handler has no role, so it never appears in the list of buttons the user can tab to; a form field whose `<label>` is not programmatically associated leaves the user to guess what the input expects. None of these defects is visible on screen. All of them are decided in code.

Assistive technology is broader than screen readers. Users with motor impairments may navigate entirely by keyboard, by switch access, or by voice control, all of which depend on focus order and on controls being reachable and large enough to target. Users with low vision magnify the page or override its colours, which breaks layouts that encode meaning through position or through colour alone. Users with cognitive disabilities depend on consistent navigation and on error messages that say how to recover. And the same properties serve users in situational difficulty: captions in a noisy room, sufficient contrast in direct sunlight, large touch targets on a moving bus.

This is why accessibility defects are amenable to the treatment this dissertation gives them. A missing `alt` attribute, an unlabelled control, or an insufficient contrast ratio is at once a lived barrier and a statically checkable property of a source file—which is precisely what makes automated detection possible, and what makes the automated *repair* of some of them plausible. The limit of that treatment, developed in Chapter 7, is that a scanner can verify that alternative text *exists* but not that it is *useful*.

A note on terminology. This dissertation follows the language of the *Lei Brasileira de Inclusão da Pessoa com Deficiência* [8] and of the UN Convention on the Rights of Persons with Disabilities, both of which use person-first constructions: *pessoa com deficiência*, "person with a disability". Earlier Brazilian legislation and technical literature used terms such as *portador de deficiência* and *pessoa com necessidades especiais*; these are no longer current, and are avoided here. The wording matters for a reason internal to the argument as well: a disability is understood as an interaction between a person and a barrier, not as a property of the person, and it is the barrier—in this case, the code—that this work sets out to remove.

2.1.2 Accessibility as a Quality Attribute

Accessibility is formally recognised by the ISO/IEC 25010 quality model as a subcharacteristic of usability, defined as the degree to which a product can be used by people with the widest range of characteristics and capabilities. In practice,

its scope encompasses users with visual, auditory, motor, and cognitive disabilities, as well as situational limitations such as bright sunlight, one-handed device use, or slow network connections. In numerous jurisdictions, accessibility carries enforceable legal obligations that extend beyond a mere quality preference [2, 8]: non-compliant digital products expose organisations to civil liability, procurement disqualification, and reputational harm.

2.1.3 The WCAG Framework

The consolidated normative reference for web accessibility is the *Web Content Accessibility Guidelines* (WCAG), published by the World Wide Web Consortium (W3C) and currently at version 2.1 [20]. WCAG organises its requirements around four foundational principles, known by the acronym **POUR**:

Perceivable Information and interface components must be presentable to users in ways they can perceive. Content that is visible only through one sensory channel (e.g., colour alone, audio without captions) violates this principle.

Operable Interface components and navigation must be operable by all users. A page that cannot be navigated with a keyboard, or that contains elements that are not reachable by assistive technologies, violates this principle.

Understandable Information and operation of the interface must be understandable. Ambiguous labels, inconsistent navigation, and error messages without correction guidance violate this principle.

Robust Content must be robust enough to be interpreted reliably by a wide variety of user agents, including assistive technologies. Malformed HTML, incorrect ARIA usage, and proprietary component implementations that bypass standard DOM semantics violate this principle.

Each principle is further broken down into *guidelines* and *success criteria*. WCAG 2.1 defines 78 success criteria at Level AA, the conformance level required by most legal frameworks and adopted as the target throughout this dissertation. Each criterion is testable: it has a specific, observable definition of pass and fail, and it is associated with one or more sufficient techniques.

2.1.4 ARIA and Semantic HTML

The Accessible Rich Internet Applications (**ARIA**) specification, maintained alongside WCAG by the W3C, provides a vocabulary of roles, states, and properties that authors can add to HTML elements to convey semantic information to assistive

technologies [21]. ARIA is not a replacement for semantic HTML; the first rule of ARIA use is to prefer native HTML elements with built-in semantics wherever possible. ARIA fills the gap for interactive patterns (sliders, tab panels, disclosure widgets, live regions) that have no native HTML equivalent. Incorrect ARIA usage (invalid roles, missing required states, attributes that contradict native semantics) is worse than no ARIA, because it actively misleads assistive technologies. Understanding this distinction is essential to appreciating why automated repair of ARIA violations is semantically complex.

2.1.5 Violation Prevalence and the Automated Detection Paradox

Despite the availability of mature automated detection tools, the WebAIM Million report (2024 edition) documents that 95.9% of the top one million websites contain detectable WCAG violations [2]. This figure has remained above 95% across every annual edition of the report since its inception, pointing to a structural problem rather than a transient knowledge gap. The paradox is that detection is not the bottleneck: tools such as Pa11y [3], Axe [4], and Lighthouse [5] can algorithmically verify roughly 30–50% of all WCAG success criteria with high precision, and they are already integrated into mainstream development workflows. (Lighthouse is named here only as a representative of this ecosystem; the framework's own four-tool detection layer is described in Section 2.2.)

The most frequent violation categories in React codebases cluster into seven operationally distinct types: missing image alternative text (`alt-text`), incorrect use of ARIA attributes (`aria`), absent form control labels (`label`), insufficient text-background contrast (`contrast`), malformed heading hierarchy (`semantic`), inadequate focus management (`focus`), and non-descriptive link text (`label`, link variant). This taxonomy, detailed in Appendix A, directly informs the dataset construction described in Chapter 5.

2.2 Automated Accessibility Detection Tools

Understanding how detection tools work is essential to understanding why their output is structurally insufficient for automated repair. The four tools integrated into *a11y-autofix* operate through distinct mechanisms that provide complementary coverage, spanning both dynamic runtime analysis and static source-level inspection.

2.2.1 Pa11y

Pa11y [3] is a command-line accessibility testing tool that drives a headless Chromium browser, loads the target page or component, and runs the **HTML_CodeSniffer** rule engine against the rendered DOM. HTML_CodeSniffer implements rule sets corresponding to WCAG 2.0, WCAG 2.1, and Section 508, each composed of dozens of individual checks. Pa11y outputs violations as structured JSON objects containing four fields: a `code` (the WCAG guideline identifier, e.g., `WCAG2AA.Principle1.Guideline1_1.1_1_1.H37`), a human-readable `message`, the `selector` (a CSS path to the violating element in the rendered DOM), and a `context` snippet (the raw HTML of the element). The WCAG guideline code encodes the principle number, guideline number, success criterion number, and the specific technique being checked, providing a direct mapping to the normative WCAG specification.

The key operational characteristic of Pa11y for this research is that it operates on the *fully rendered DOM* (after JavaScript execution and CSS application). This is important for React applications, where the component tree is constructed dynamically at runtime. Pa11y therefore captures violations that would be invisible to a static HTML parser.

2.2.2 Axe-Core

Axe-core [4] is a JavaScript accessibility engine developed by Deque Systems that evaluates the live DOM against a curated set of accessibility rules. Its central design philosophy is **zero false positives**: every rule is only triggered when the engine is statistically certain the element fails accessibility requirements, accepting reduced recall in exchange for high precision. This design choice has made axe-core the de facto standard for automated accessibility testing in CI/CD pipelines.

Axe-core enriches its output beyond what Pa11y provides: each violation includes an `impact` severity (critical, serious, moderate, or minor), a `help` URL linking to detailed repair guidance, and an array of `nodes` each with the element's HTML, a CSS selector, and a human-readable failure summary. It also provides a rule identifier (e.g., `color-contrast`, `image-alt`) that is more stable and searchable than WCAG guideline codes.

In *a11y-autofix*, axe-core is invoked in two modes: directly via a Node.js subprocess for fast static scanning, and through Playwright for dynamic evaluation of rendered React components (see below). The zero-false-positive property is exploited in the confidence assignment mechanism: a critical or serious finding from axe-core alone is treated as MEDIUM confidence even without cross-tool agreement, because the probability of a false positive from axe-core at those severity levels is

empirically very low.

2.2.3 Playwright with Axe-Core

React applications render their component trees dynamically: the DOM state after JavaScript execution and React's reconciliation process may differ substantially from any static snapshot. Components that initialise state on mount, conditionally render content, or apply CSS-in-JS styles at runtime will not be fully represented by a static scan. Playwright [22] is a browser automation framework that provides programmatic control over a full browser instance, enabling evaluation of the component at any point in its lifecycle.

In the *a11y-autofix* scanning pipeline, Playwright is configured to: (1) launch an isolated Chromium browser instance; (2) load the React component file in a sandboxed test harness that renders it with default props; (3) wait for all mount effects and asynchronous state updates to settle; and (4) inject the axe-core engine into the page and execute a full scan of the rendered output. This two-step process, browser automation followed by axe-core injection, captures a class of violations that neither Pa11y nor direct axe-core invocation can detect in isolation, particularly violations that only manifest after user interaction or asynchronous data loading.

2.2.4 ESLint with the jsx-a11y Plugin

The three dynamic scanners described above share a fundamental limitation: they require the component to be rendered inside a browser environment before any analysis can occur. This dependency means they are blind to violations that are structurally encoded in the source code but that a particular test harness does not trigger: for example, a missing `label` association on an `<input>` that only appears when a specific prop is passed, or a keyboard event handler absent from a conditionally rendered element.

ESLint [23], the dominant static analysis tool for JavaScript ecosystems, addresses this gap through *compile-time inspection* of the JSX abstract syntax tree. When configured with the `eslint-plugin-jsx-a11y` [23] plugin, ESLint traverses the JSX node tree and applies a set of accessibility rules without executing the component. The 21 rules activated in *a11y-autofix* cover all major WCAG 2.1 Level AA categories addressable through static analysis: missing `alt` attributes on `<img>` elements (`jsx-a11y/alt-text`, WCAG 1.1.1), absent or malformed label associations on form controls (`jsx-a11y/label-has-associated-control`, WCAG 1.3.1), click events without corresponding keyboard handlers (`jsx-a11y/click-events-have-key-events`, WCAG 2.1.1), interactive elements missing focus management (`jsx-a11y/interactive-supports-focus`,

WCAG 2.1.1), positive `tabIndex` values that disrupt natural focus order (`jsx-a11y/tabindex-no-positive`, WCAG 2.4.3), invalid or unsupported ARIA roles and properties (`jsx-a11y/aria-role`, `jsx-a11y/aria-props`, WCAG 4.1.2), and anchor elements without accessible content (`jsx-a11y/anchor-has-content`, WCAG 4.1.2).

An important operational characteristic of this scanner is that it produces findings expressed in terms of *source file coordinates* (file, line, column) rather than CSS selectors into a rendered DOM. This makes its output directly actionable for patch generation: the selector produced by the ESLint runner is a `filename:line:column` reference that the Babel AST traversal can resolve to an exact JSX node without any selector-to-source mapping step. This is the converse of the problem faced by Pa11y and Axe-core, whose DOM selectors must be mapped back to source locations through AST analysis because the rendered DOM has lost the component boundary information present in the original TSX.

The ESLint runner in *a11y-autofix* handles compatibility across major ESLint versions. ESLint 8 and earlier use a legacy `.eslintrc.json` configuration format; ESLint 9 and later adopted a flat configuration model (`eslint.config.cjs`) that differs structurally. The runner detects the installed ESLint major version at initialisation time and writes an appropriate temporary configuration file, resolving plugins from the global npm registry via `NODE_PATH`. This design allows the framework to analyse any React/TypeScript repository without requiring that project to have ESLint or `jsx-a11y` installed locally, which would be a prohibitive constraint when scanning the broad and heterogeneous set of repositories in the benchmark.

2.2.5 Multi-Tool Consensus and Its Significance

Prior research has consistently shown that individual accessibility scanners have non-trivial false positive and false negative rates, and that different tools detect distinct subsets of violations on the same pages [24, 25]. Running four independent scanners and comparing their outputs exploits this complementarity: a violation reported by two or more tools is far more likely to be genuine than one reported by a single tool, because independent tools with different heuristics and rule engines would need to fail in the same direction simultaneously. The consensus is particularly meaningful across the dynamic–static boundary: when Pa11y (runtime), Axe-core (runtime), and ESLint (compile-time) all flag the same element, the joint probability of a false positive is negligible. This *consensus principle* is the foundation of the confidence scoring mechanism described in Section 4.1 and directly influences which violations are annotated automatically versus requiring human review.

2.3 The Detection–Correction Gap

With detection tool mechanics established, this section names the gap precisely: the structural mismatch between what scanners produce and what a repair system needs.

2.3.1 Structure of Detection Tool Output

The asymmetry between automated detection and manual correction constitutes what this dissertation terms the *detection–correction gap*. To understand the gap structurally, consider a representative Pa11y output for a link accessibility violation:

```
{
  "code":     "WCAG2AA.Principle2.Guideline2_4.2_4_4.H77",
  "message":  "Check that the link text identifies the purpose of the link",
  "selector": "main > section:nth-child(3) > a:nth-child(2)",
  "context":  "<a href=\"/products\">Read more</a>"
}
```

This output communicates *what* is wrong (a link fails WCAG 2.4.4) and *where* it is (a CSS path into the rendered DOM). It communicates nothing about: what the link's destination contains or implies about an appropriate label; what React component renders this element and what props it receives; what naming conventions the project uses for accessible labels; whether the link text is hard-coded or dynamic; or what the surrounding heading context is, which WCAG 2.4.4 explicitly allows to contribute to link purpose. A developer with domain knowledge can resolve all of these questions by reading the source code; an automated system receiving only this JSON cannot.

2.3.2 Why Existing Auto-Fix Tools Are Insufficient

Existing rule-based auto-fix tools (the `-fix` flag in ESLint with the `jsx-a11y` plugin [23], or Prettier's formatting corrections) address only transformations that are fully determined by the violation type and the element's local syntax, with no dependence on surrounding context. Adding a mandatory `alt` attribute with an empty string to an `<img>` element is such a transformation: it is always syntactically valid, and whether it is semantically correct (decorative image) or incorrect (informative image) is a question that rule-based tools sidestep by simply applying the minimal compliant change. Violations that require semantic reasoning, like choosing a descriptive `alt` text that accurately conveys the image's content, or determining that

a link's accessible name should incorporate its surrounding heading, are outside the scope of any rule-based system.

The practical cost of the gap compounds in real projects. Each violation requires the developer to: (1) interpret the report and locate the element in the source code; (2) research the applicable WCAG criterion and its sufficient techniques; (3) design a fix compatible with the framework and project conventions; and (4) implement and validate the change without introducing regressions. In medium-sized codebases with dozens of recurring violations, this process consumes a disproportionate share of maintenance time.

2.4 Large Language Models for Code Generation

2.4.1 Why a Language Model, and Not Something Else

Before describing what LLMs are, it is worth stating why the repair step in this dissertation is delegated to one at all. Automated program repair has a long history that owes nothing to language models, and three of its established families were considered and set aside for reasons specific to the accessibility domain.

Rule-based transformation—the approach of ESLint's `-fix`—is the natural first candidate and remains the right tool wherever the correct output is a function of the violation type alone. Its ceiling is the one described in Section 2.3: it can insert `alt=""` but cannot decide whether the image is decorative, because that decision requires information no rule can encode. Roughly half the corpus used in this dissertation consists of violations of exactly this kind.

Search-based repair, in the tradition of GenProg and its successors [26, 27], mutates a program and keeps the variants that pass a test suite. It depends on a fitness signal dense enough to guide the search. Accessibility offers no such signal: a component either satisfies a scanner rule or does not, the search space of edits is enormous, and there is no partial credit to climb. The oracle is a gate, not a gradient.

Learning-based repair trained on defect–fix pairs would be viable given a large labelled corpus of accessibility defects and their corrections. No such corpus exists at the scale supervised training requires—which is itself part of the problem this dissertation addresses, and the reason its own benchmark comprises 168 violations rather than tens of thousands.

What makes an LLM the fitting instrument is that the knowledge the other approaches lack has already been absorbed during pre-training. WCAG techniques, ARIA patterns, and React idioms are extensively documented on the public web, so a code-specialised model has read them without ever being trained on a labelled repair

dataset. The task then stops being one of *acquiring* domain knowledge and becomes one of *eliciting and constraining* it, which is the conjecture stated in Section 1.2 and tested in Chapter 6. The cost of this choice is a generator that is non-deterministic and confidently wrong in characteristic ways—the failure modes of Section 2.4.5—and it is that cost which motivates the validation architecture of Chapter 4.

2.4.2 Architecture and Pre-training

Large Language Models (LLMs) are neural networks based on the Transformer architecture [28], trained to predict the next token in a sequence over massive corpora of text and code. Pre-training on code-centric datasets (GitHub repositories, Stack Overflow, documentation, and technical specifications) endows these models with implicit knowledge of programming languages, framework idioms, and domain conventions. Models such as Codex [11], CodeLlama [12], Qwen2.5-Coder [13], and DeepSeek-Coder [14] are fine-tuned specifically on code corpora, improving their ability to generate, complete, and transform code relative to general-purpose models.

The key property that makes LLMs relevant to this research is in-context generalisation: given a sufficiently descriptive prompt, the model can perform tasks it was never explicitly trained on, by activating and recombining patterns learned during pre-training. An LLM that has seen thousands of React components during training has implicitly learned the relationship between ARIA roles and component patterns, even if it was never trained on a dataset labelled "accessibility violations and their fixes".

2.4.3 Local Deployment and Privacy

A critical design constraint of *a11y-autofix* is that all LLM inference runs locally, on open-weight models, without sending source code to external APIs. This is achieved through **Ollama**, a runtime for locally serving quantised LLM weights on commodity hardware. Ollama exposes a REST API identical in structure to OpenAI's `/v1/chat/completions` endpoint, allowing the framework to switch between models by changing a configuration parameter.

The constraint is deliberate, and it costs something: the frontier proprietary models would almost certainly repair more violations than a quantised 7B model running in 6 GB of VRAM. Four reasons make the trade worthwhile for this work.

The first is **deployability under the conditions the problem actually occurs in**. The organisations facing the compliance exposure described in Chapter 1 include public bodies and companies whose source code cannot leave their infrastructure, by policy or by contract. A repair tool that requires uploading proprietary

source to a third-party endpoint is unavailable to precisely the users the LBI obliges to act. Local inference removes that barrier, and doing so on commodity hardware—a single consumer GPU rather than a datacentre—keeps the result within reach of a small team.

The second is **reproducibility**. A proprietary endpoint is a moving target: the weights behind a model name change without notice, deprecations retire versions, and a result reported today may be unreproducible next year through no fault of the experimental design. Open weights can be pinned by version and digest, downloaded by anyone, and re-run indefinitely. For a dissertation whose contribution includes a benchmark meant to be reused, this is not a secondary consideration.

The third is **cost as an experimental constraint**. The evaluation reported here runs three models over 1,249 files in three repetitions, plus two ablations—tens of thousands of inference calls. Under per-token API pricing, that design would have been shaped by budget rather than by methodology.

The fourth is that a local, open-weight baseline is **the conservative claim**. Results obtained on a 3B model in 6 GB establish a floor: whatever a frontier model would achieve, it would not achieve less. A study reporting that GPT-class models repair some fraction of violations would leave open whether the result depends on scale unavailable to most practitioners. This one does not.

What the constraint does *not* license is any inference about performance outside the 3B–7B quantised band actually tested, a limit stated wherever model-comparison results are reported (Section 5.7).

2.4.4 Hallucination as a Structural Property

The generative mechanism that makes an LLM useful for repair also produces its most characteristic defect, and the two cannot be separated. A language model does not retrieve facts; it samples the next token from a distribution conditioned on everything before it. Nothing in that procedure distinguishes a sequence that is well attested in the training data from one that merely *resembles* well-attested sequences. *Hallucination* is the name for the second case: fluent, syntactically valid output asserting something that is not true of the world it describes.

Three properties of the phenomenon matter for automated repair.

It is **not an error state**. The model does not fail, warn, or lower a confidence score when it hallucinates; the output is produced by the same process, with the same fluency, as a correct answer. There is no internal signal a caller can read to tell the two apart, which is why detection must be external to the model.

It is **most likely exactly where the domain is most structured**. ARIA is a closed vocabulary of roles, states, and properties whose members share

strong surface regularities—`aria-label`, `aria-labelledby`, `aria-describedby`, `role="navigation"`. A model that has learned those regularities can generate `aria-role="navigation-button"`, which obeys every visible pattern and exists nowhere in the specification. The very consistency that lets the model produce correct ARIA most of the time is what lets it manufacture plausible ARIA the rest of the time.

It is **semantic as often as it is lexical**. Inventing an attribute is the easy case: the name either appears in the specification or it does not, so a lookup catches it. The harder case is a well-formed, entirely real construct applied to a situation it does not fit—`alt="decorative image"` on a decorative image, where every token is legitimate and the result still defeats the purpose of the attribute. No vocabulary check detects this.

These three properties shape the framework's response. Because hallucination carries no internal signal, every patch is checked externally rather than trusted (Chapter 4). Because lexical hallucination is mechanically detectable, it is caught by the syntactic and domain-compliance layers. And because semantic hallucination is not mechanically detectable, it survives—which is why it appears among the residual failure modes in Chapter 7 rather than among the suppressed ones.

2.4.5 Failure Modes When Applied Naively to Repair

Feeding raw detection tool output directly to an LLM rarely produces adequate repairs. Four recurring failure modes were identified in preliminary experiments. The first and third are the lexical and semantic hallucinations just described; the other two arise from the model's tendency to edit beyond the violation it was given.

API hallucination The model generates syntactically plausible but non-existent ARIA attributes or HTML properties (e.g., `aria-role="navigation-button"`, where the correct attribute is `role` and the value `navigation-button` does not exist in the ARIA specification). This occurs because the model extrapolates from partial patterns without verifying against the normative specification.

Functional regression The model rewrites portions of the component beyond what is necessary to resolve the violation, altering event handlers, state management logic, or prop interfaces. This failure mode is particularly dangerous because it may pass automated accessibility checks while introducing subtle runtime bugs.

Specification misinterpretation The model produces a fix that satisfies the literal violation report but violates the standard's intent. A canonical example is

inserting `alt="decorative image"` instead of `alt=""` for a decorative image: the former causes screen readers to announce the phrase "decorative image" to users, which is both redundant and confusing.

Framework convention violation The model generates code that is syntactically valid but violates React's programming model, such as using `document.getElementById()` instead of the idiomatic `useRef` hook. Such code may work in the browser but breaks server-side rendering and violates the declarative paradigm that makes React components testable and reusable.

These failure modes establish the technical requirement for structured prompt engineering and multi-layer validation, both of which are core contributions of this work.

2.5 AI Agents for Automated Code Repair

2.5.1 From LLMs to Agents

A Large Language Model generates text given a prompt; an agent uses a language model as its reasoning engine within a control loop that perceives state, plans actions, executes tools, observes results, and iterates [29]. The distinction is architectural: agents can take multi-step actions, access external tools (file systems, compilers, web browsers, test runners), maintain state across steps, and revise their approach based on intermediate feedback. For code repair tasks, the agent paradigm is essential because a single LLM call is generally insufficient: the model needs to read the source file, understand the violation, generate a patch, validate the patch against a compiler and a set of tests, and potentially revise based on validation failures.

The ReAct pattern [29] (Reasoning + Acting) is the dominant paradigm for LLM agents. In ReAct, the model alternates between generating a *thought* (an internal reasoning step) and an *action* (an external tool invocation), observing the tool's output before proceeding. This pattern allows the agent to decompose complex repairs into smaller steps, verify intermediate results, and recover from errors, none of which are possible in single-pass generation.

2.5.2 Multi-Agent Architectures

Complex repair tasks benefit from *multi-agent architectures*, in which specialised agents with distinct roles and capabilities collaborate under an orchestrator [30]. The **hierarchical planner–executor** pattern is the most common: a planner agent

decomposes the task and delegates subtasks to executor agents, each of which uses tools appropriate to its specialisation. This separation of concerns mirrors software engineering team structures and has been shown to reduce error rates relative to a single-agent approach for moderately complex tasks.

a11y-autofix borrows from this idea at the level of *routing* rather than of concurrent agent execution: a two-tier router places each file on the complexity spectrum. A **SWE-agent**-style engine [15] handles the common case of surgical, line-targeted edits, while an **OpenHands**-style sandboxed environment [16], with shell and file-system access, is reserved for violations that require reading related files or modifying design tokens. Both paths are driven by the same local LLM through Ollama; the router selects between them per file using a scoring matrix over violation characteristics.

The scope of this borrowing belongs here rather than later, since it qualifies how the framework should be read for the remainder of the dissertation. In the execution environment used for the evaluation, neither the SWE-agent nor the OpenHands command-line tools were installed. Both routing tiers therefore resolved to the same prompt-based adapter (`DirectLLMAgent`); what distinguishes them is the *format* of the prompt they assemble, while the underlying repair mechanism is identical. No autonomous agent loop and no concurrent collaboration between specialised agents was exercised in any reported experiment. The framework is accordingly described throughout as a four-phase repair pipeline with adaptive routing rather than as a multi-agent system. Section 4.4 details the full routing logic and this operationalisation.

2.5.3 Iterative Repair and Feedback Loops

A key advantage of the agent paradigm over single-pass generation is the ability to iterate based on validation feedback. Rather than accepting or rejecting a patch in a single step, it is possible to pass structured failure diagnostics back to the repair agent, which can then revise its approach. This *feedback-driven retry* mechanism converts validation failures from terminal rejections into learning signals. Research on autonomous repair agents such as RepairAgent [31] and AutoCodeRover [32] consistently demonstrates that iterative approaches outperform single-pass generation, particularly for violations where the first repair attempt reveals information about the codebase that was absent from the initial prompt.

2.6 Prompt Engineering for Code Repair

2.6.1 The Role of Prompt Engineering

Prompt engineering is the practice of designing and structuring the natural language input to an LLM to elicit accurate, well-formed, and task-appropriate output. For code generation and repair, prompt quality is a first-order determinant of output quality: the same model can produce dramatically different outputs depending on how the task is specified [33]. Unlike model fine-tuning, which modifies model weights, prompt engineering operates entirely at inference time, making it the accessible adaptation mechanism for practitioners who cannot retrain foundation models.

2.6.2 Core Prompting Strategies

Three prompting strategies are evaluated in this dissertation and require precise definition. Zero-shot prompting provides a task description without any input–output examples, requiring the model to rely entirely on knowledge encoded during pretraining to infer the expected output format and quality; it is the simplest and cheapest strategy but typically produces lower accuracy on specialised tasks [34]. Few-shot prompting prepends k input–output demonstration pairs to the prompt before the actual task instance [34], activating relevant knowledge patterns and implicitly communicating output format, repair style, and domain conventions. Research consistently shows that even two to three high-quality examples produce substantial accuracy improvements, and the *selection* of examples matters as much as their presence: semantically similar examples outperform randomly chosen ones [35]. In *a11y-autofix*, examples are retrieved by cosine similarity between the current violation's embedding and a pre-built library of violation–fix pairs, a technique known as *retrieval-augmented few-shot learning*. Chain-of-thought (CoT) prompting instructs the model to generate an explicit step-by-step reasoning trace before producing its final answer [36], reducing errors on tasks that require multi-step inference by externalising the model's reasoning process. For code repair, CoT encourages the model to identify the root cause before generating a fix, reducing surface fixes that resolve the symptom without addressing the underlying problem; its cost is increased token consumption and generation latency.

2.6.3 Structured Prompts for Code Repair

For code repair specifically, the literature identifies a set of prompt components that must operate synergistically to achieve high-quality output [37, 38], and the absence of any one component activates a corresponding failure mode. A *role defi-*

nition establishes the model's operational persona (e.g., "expert React accessibility engineer"), activating domain-relevant knowledge patterns from pre-training and priming conservative, precise generation behaviour. A *violation context* block transforms the raw scanner output into a semantically rich description of the accessibility barrier, including the full WCAG criterion text, its rationale, and its real-world impact on users, compensating for the sparse, code-centric nature of scanner reports. A *code context* block provides the source file to be modified, together with its prop interface, imported modules, and any relevant surrounding components; without this context, the model cannot determine whether a proposed fix is compatible with the component's architecture. *Explicit constraints* enumerate the boundaries the model must respect: minimal change, functional preservation, framework conventions, and API existence, each targeting a specific failure mode from Section 2.4. *Few-shot exemplars* demonstrate the expected repair pattern through concrete violation–fix pairs selected for semantic relevance to the current instance. Finally, an *output format specification* defines the exact response schema, enabling automated parsing and validation without heuristic extraction. The six-component template adopted by *a11y-autofix* instantiates this full set; its detailed specification and assembly algorithm are presented in Section 4.3.

2.6.4 Ablation Studies

A design assembled from several parts (the six-component template above, or the three prompting strategies) invites an obvious question: how much does each part actually contribute? The standard tool for answering it is the *ablation study*, a term borrowed from neuroscience, where a brain region's function is inferred by removing it and observing the resulting deficit. Machine-learning research adopted the same logic to attribute a system's performance to its parts, switching each component off (or systematically varying it) and measuring how much the result changes [39, 40]. The method is only as trustworthy as its controls: everything except the component under test is held fixed, which makes an ablation a one-factor-at-a-time experiment in the classical sense [41]. Two cautions follow and matter for reading any ablation. First, it reveals contribution *within a given system*, not absolute importance: a part that looks inert beside a strong baseline may be decisive in another configuration. Second, a non-significant result is not proof of irrelevance, since with few independent observations a real effect can simply go undetected, so an ablation's null findings must be read together with its statistical power. This dissertation applies the method twice (Chapter 5): once to choose among prompting strategies, and once to weigh the framework's own structural components.

2.7 React and TypeScript as the Target Platform

a11y-autofix targets React/TypeScript codebases specifically. This scoping decision warrants brief justification and introduces terminology used throughout the dissertation.

React is a declarative JavaScript library for building user interfaces through composable *components* (functions or classes that accept *props* (input parameters) and return JSX, a syntax extension that describes the rendered output). React's virtual DOM model means that direct DOM manipulation (e.g., `document.getElementById`) is an anti-pattern: components describe *what* to render, not *how* to manipulate the DOM. This declarative model has implications for accessibility repair: fixes must respect React's unidirectional data flow, hook rules, and event delegation model. React applications account for the largest share of modern front-end codebases [42], making them the highest-value target for accessibility automation research.

TypeScript is a statically typed superset of JavaScript that compiles to plain JavaScript. Its type system enforces component prop interfaces at compile time, making it possible to detect structural regressions (such as removing a required prop) without executing the application. The framework exploits the TypeScript compiler (`tsc -noEmit`) as one of its four validation layers. JSX is the syntax extension used in both React JavaScript and React TypeScript (`.jsx`/`.tsx`) files, allowing HTML-like markup to be embedded directly in component functions; JSX elements map to React API calls at compile time, and understanding JSX well-formedness is necessary for the syntactic validation layer of the pipeline.

The combination of React's component model, TypeScript's static typing, and JSX's HTML-like syntax creates a repair target that is richer in structure than plain HTML but more constrained than arbitrary JavaScript, making it particularly amenable to LLM-based repair with structured validation.

The six conceptual blocks established in this chapter define the design space for *a11y-autofix*. Chapter 3 examines how the research community has answered analogous questions in adjacent domains, providing the empirical basis for the specific design decisions made in Chapter 4.

2.8 Summary

This chapter established the conceptual chain the rest of the dissertation depends on.

Accessibility was presented first as an experience before being treated as a property of code: what a screen reader actually reads, why a missing `alt` attribute

is a barrier rather than a formatting lapse, and why those barriers are nonetheless decided in markup. That dual nature—lived consequence, statically checkable property—is what makes automated detection possible and automated repair conceivable. WCAG 2.1 supplies the normative vocabulary, and the scanners integrated by the framework supply the detection, but their reports were shown to be structurally sufficient for diagnosis and insufficient for repair. That deficiency is the detection–correction gap.

The chapter then argued why a language model is the fitting instrument for closing it, by elimination rather than by enthusiasm: rule-based transformation cannot decide what a rule does not encode, search-based repair has no fitness signal to climb in this domain, and learning-based repair has no labelled corpus at the necessary scale. What an LLM contributes is knowledge already absorbed during pre-training, which reframes the task from acquiring domain knowledge to eliciting and constraining it. Running that model locally, on open weights, was justified on grounds of deployability, reproducibility, experimental cost, and conservatism of the resulting claim.

The same generative mechanism was shown to carry a characteristic defect. Hallucination is not an error state, is most likely where a domain is most regular, and appears in a lexical form that an external reference can decide and a semantic form that it cannot. That asymmetry, together with the four naive failure modes catalogued in Section 2.4.5, establishes the requirement that every generated patch be checked externally rather than trusted—the requirement the framework of Chapter 4 is built to satisfy.

Chapter 3 turns from concepts to evidence, examining what the published literature reports about agentic code repair and where it is silent.

Chapter 3

Rapid Review: Artificial Intelligence Agents for Code Repair and Generation

This chapter presents a structured Rapid Review of the literature on AI-agent-based code repair and generation (2020–early 2026), conducted to *inform* the architectural and experimental decisions of the *a11y-autofix* framework. Its role is to establish what the field converges on and where it is silent, so that the design choices that follow are made deliberately and against a documented backdrop; it is not offered as confirmatory evidence that those choices are correct, and the reduced-rigour concessions inherent to the rapid-review format—documented in Section 3.6—bound how much weight its conclusions can carry. Section 3.1 motivates and scopes the review. Section 3.2 describes the search and selection protocol in full (the complete protocol artefacts are also available in Appendix B). Section 3.3 reports the selection outcomes. Section 3.4 synthesises the findings across architectural patterns, prompting strategies, validation mechanisms, and open research gaps. Section 3.5 discusses the implications of the review for the framework design.

3.1 Motivation and Scope

Chapter 2 established why the detection–correction gap exists and identified the conceptual components of a solution: multi-agent architectures, structured prompt engineering, and layered validation. Before committing to specific design choices for *a11y-autofix*, this chapter asks a prior question: what does the research community already know about building systems like this? The Rapid Review synthesises 55 peer-reviewed studies (2020–early 2026) on AI-agent-based code repair in order to answer it within the limits of the format. Its purpose is not to survey the field for its

own sake, nor to validate the design bets made in Chapter 2—a review of this scope cannot do that—but to situate those bets against the published record and to surface the five specific gaps that this dissertation's contributions address. Section 3.5 maps each finding directly to an *a11y-autofix* design decision; the reader who wants the short version can start there and return to the synthesis for supporting evidence.

The review covers: (1) agentic systems for code repair and transformation; (2) multi-agent architectures applied to software engineering tasks; (3) prompt engineering and context-injection strategies for code repair with LLMs; and (4) repair validation mechanisms used as oracles for correctness and quality. It focuses on *agentic* approaches: systems where an AI component exhibits autonomy via planning, tool use, iterative execution, or explicit multi-agent collaboration. Pure next-token generation studies without an agentic control loop are included only when they provide methodological insights critical to agent design (e.g., validation, prompting, or benchmarking).

3.2 Review Protocol

3.2.1 Research Questions

The protocol is organised around the following research questions (RQs):

RQ1 What AI-agent-based approaches have been proposed for code repair, correction, refactoring, or transformation, and what are their architectural foundations?

RQ2 How are multi-agent systems structured and orchestrated for software engineering tasks, especially code repair?

RQ3 Which prompt engineering and context construction strategies are reported as effective for LLM-driven repair and code transformation?

RQ4 Which validation mechanisms are used to ensure correctness and quality of automatically generated repairs?

RQ5 What limitations and research gaps are explicitly identified, and which gaps motivate new contributions?

3.2.2 Eligibility Criteria

Eligibility criteria were defined *a priori* to bound scope and reduce selection drift during screening.

Inclusion criteria (IC).

IC1 Studies that propose, evaluate, or analyse AI agents (including LLM-based agents) for code repair, bug fixing, refactoring, code transformation, or code generation with an explicit agentic control loop.

IC2 Publication year between 2020 and 2026 (inclusive), reflecting the emergence and consolidation of large-scale LLMs for code.

IC3 Studies indexed in Scopus under `COMP` or `ENGI`.

IC4 Peer-reviewed venues (conference/journal), book chapters, and review papers written in English.

Exclusion criteria (EC).

EC1 Proceedings-only records without individually attributable papers.

EC2 Duplicates; only the most complete or most recent version is kept.

EC3 Studies unrelated to code repair/transformation/generation by AI agents (e.g., robotics agents, navigation agents, chemical "agents", or non-code ML applications).

3.2.3 Search Strategy

The search was executed in **Scopus**. The query was constructed using a PICO-inspired structure adapted to software engineering reviews [43], with three conceptual blocks: (i) AI/ML intervention, (ii) code repair/transformation outcomes, and (iii) agentic architecture as a differentiator.

```
TITLE-ABS-KEY(
  (
    ("artificial intelligence" OR "AI" OR "machine learning"
     OR "deep learning" OR "neural network*")
    AND
    ("code repair" OR "code correction" OR "code modification"
     OR "code fixing" OR "bug fixing" OR "automated repair"
     OR "program repair" OR "code refactoring"
     OR "code generation" OR "code transformation")
    AND
    ("agent*" OR "autonomous agent*" OR "AI agent*"
     OR "intelligent agent*" OR "software agent*")
  )
)
```

```
AND PUBYEAR > 2019
AND (
  LIMIT-TO(SUBJAREA,"COMP")
  OR LIMIT-TO(SUBJAREA,"ENGI")
)
```

The query intentionally privileges recall over precision within an agentic frame: the term `agent*` can retrieve irrelevant meanings (network agents, chemical agents). This is expected and handled via explicit exclusion during screening. Conversely, terms such as "copilot" or "code assistant" are not explicitly included; this is treated as a threat to construct validity (Section 3.6) and partially mitigated by broad AI terminology.

3.3 Selection Results

The Scopus query returned **138 records** after the automated filters for publication year and subject area. Screening outcomes are summarised in Table 3.1.

Table 3.1: Rapid Review screening summary

Stage	**Records**	**Cumulative excluded**
Records retrieved from Scopus search	138	–
Duplicate removal	18	18
Records after duplicate removal	120	18
Excluded in title/abstract screening	63	81
Excluded in full-text screening	2	83
Studies included in synthesis	**55**	–

Title/abstract exclusions were dominated by: (i) "agent" used in non-AI meanings (e.g., networking, chemistry); (ii) AI applications unrelated to code; (iii) HDL-only work with limited transfer; and (iv) incomplete records without an attributable paper.

The 55 included studies are strongly concentrated in 2024–2025, consistent with the rapid expansion of LLM-for-code research: a single study each from 2021 and 2023, 23 studies from 2024 (41.8%), 26 from 2025 (47.3%), and 4 from 2026 (7.3%).

Based on Scopus record types for included studies, the corpus is dominated by conference publications: **40 conference papers** (72.7%), **12 journal articles** (21.8%), and **3 book-oriented publications** (books or book chapters; 5.5%).

3.4 Synthesis of Evidence

3.4.1 Synthesis Procedure

Because the credibility of a thematic synthesis rests on how the codes were produced, the procedure is stated here in full.

The thematic coding was performed by the author of this dissertation, working alone, on the 55 included studies. The unit of analysis was the individual study; the evidence base for each was its title, abstract, keywords, and the portions of the full text describing architecture, prompting, and validation. The thematic coding scheme was not defined a priori. It was built inductively in two passes: a first open-coding pass over the corpus produced a flat list of descriptive labels for the mechanism each study contributed (for instance PLANNER–EXECUTOR SPLIT, TEST-SUITE ORACLE, RETRIEVAL-AUGMENTED EXEMPLARS); a second pass grouped those labels into the higher-level categories reported in Sections 3.4.2–3.4.6, merging labels that proved to be the same mechanism under different names and discarding labels that carried only a single study and no connection to the review questions. The five reported clusters correspond to the five protocol research questions, so the mapping from label to cluster was constrained by the protocol rather than chosen freely after the fact.

Neither the thematic coding scheme nor the study-to-category assignment was reviewed by a second researcher. There was no independent second round of thematic coding and no reconciliation step, so no inter-rater agreement statistic such as Cohen's κ can be reported: none is defined for a single coder. Nor was there an adjudication protocol, since a single coder generates no disagreements to adjudicate. That silence follows from how the process was set up and should not be read as evidence that it ran smoothly.

This is the principal methodological weakness of the review, and its consequence is concrete. The boundaries between the reported categories—above all the boundary that produces the four architectural patterns and the five gaps—rest on one analyst's unchecked judgement. A second coder could plausibly have drawn them elsewhere, so neither the categories nor the counts derived from them carry the reproducibility that a double-coded synthesis would license. The categories also describe the retrieved corpus and make no claim to exhaustiveness over the field.

Two features of the procedure mitigate the weakness without removing it. The five protocol research questions were fixed before thematic coding began and constrained the mapping from label to cluster, which limits how freely the categories could be chosen after the fact. And Appendix B.5 enumerates the per-study assignment, so a reader who disagrees with a category boundary can see which studies move and re-derive the counts. The review did not control this threat, but it left

it open to audit. Section 3.6 carries it forward, and Section 8.4 lists independent re-coding of the corpus among the review's extensions.

The synthesis that follows emphasises: (i) what the field appears to converge on; (ii) where evidence is still weak or missing; and (iii) which design implications follow for an automated accessibility repair system.

3.4.2 RQ1: Agentic Systems for Code Repair and Transformation

From classical APR to autonomous LLM repair agents

Across the corpus, the dominant shift is from *Automated Program Repair* (APR) based on hand-designed templates and heuristic search [27] towards **LLM-centred agents** that operate in iterative perceive–plan–act loops. In these systems, the LLM is not merely a patch generator but a control policy that selects tools (e.g., file inspection, test execution, patch application) and revises hypotheses over multiple cycles.

RepairAgent [31] is the most directly aligned to the goals of this dissertation: an explicitly autonomous LLM-based repair agent that uses tools and iterative repair attempts. Its central claim is architectural: multi-step autonomy is required for bugs that demand interdependent actions (localisation, patching, and validation across files).

AutoCodeRover [32] integrates issue retrieval, fault localisation, and patch generation into an agentic pipeline. A key engineering principle is **context precision**: structural code search (e.g., AST-guided retrieval) narrows the prompt context to relevant code regions, supporting improved patch quality on SWE-bench-style tasks.

ASPIRE [44] addresses *execution-free repair* via a multi-agent design (evaluator, analyser, fixer), motivated by contexts where running the code is costly or infeasible. This assumption maps naturally to UI and accessibility repair settings, where end-to-end execution and oracle construction can be expensive, flaky, or non-deterministic.

Beyond general-purpose software, agentic repair appears in domain-constrained contexts. For example, **Guo et al.** [45] explore agentic optimisation for smart contract repair under multiple objectives (security, gas efficiency, functional correctness), illustrating a transferable idea: **repair as multi-criteria optimisation** rather than a single pass/fail outcome.

Surveys and consolidation studies

The corpus also includes consolidation efforts that are methodologically useful even when they do not introduce a new system. **Zhang et al.** [46] map LLM-based bug fixing approaches across interaction styles (single-shot, human-in-the-loop, autonomous agents) and explicitly point to open directions: runtime repair, specification-guided repair, and CI/CD integration. These directions closely mirror the practical deployment constraints faced by automated remediation tools.

A recurring implication across survey-oriented studies is that strong results often depend as much on **workflow design** (context, validation, and iteration) as on model choice [46, 47]. Key evaluation dimensions identified across systems include pass@k, functional preservation, code quality, and computational cost, underscoring that no single metric suffices to characterise repair quality.

3.4.3 RQ2: Multi-Agent Architectures for Software Engineering Tasks

Emerging architectural patterns

Four architectural patterns recur across multi-agent software engineering systems. The most common is the hierarchical planner–executor topology, in which a coordinator agent decomposes tasks and assigns subtasks to specialists. Akilesh et al. [30] instantiate this pattern for multi-step code generation workflows, and Ramirez-Rueda et al. [48] report advantages over single-agent baselines for moderately complex tasks; the practical advantage is control, since ordering constraints and tool dependencies are easier to enforce under central orchestration.

A second recurring pattern is role-based specialisation, in which agents emulate software team roles (developer, reviewer, tester, architect). Lin et al. [49] propose a Debate–Verify–Debug design where independent agent roles interact to reduce error rates, and RefactorGPT [50] applies similar role separation to refactoring, separating smell detection, refactoring proposal, and equivalence checking.

A third pattern is sequential pipelines with feedback loops, where stage outputs are validated and may trigger rework upstream. This pattern appears prominently in repair agents that iterate based on validation feedback [31, 32] and in multi-agent workflows that interleave generation and verification steps [49].

A fourth and more recent pattern is dynamic filtering through reputation mechanisms. As multi-agent systems scale, agent reliability becomes a system-level problem, and DRF [51] proposes reputation-based filtering to mitigate unreliable agents in large-scale LLM pipelines.

Orchestration frameworks as de facto infrastructure

The corpus indicates convergence on a small set of orchestration frameworks.

LangChain/LangGraph are frequently used as implementation substrates. **Mandulapalli et al.** [52] analyse agentic workflow development patterns and discuss centralised vs. decentralised coordination, with centralised orchestration generally preferred when execution order and tool constraints are critical (as in repair).

AutoGen, CrewAI, and TaskWeaver are compared by **Barbarroxa et al.** [47], who report practical trade-offs: AutoGen provides flexibility for iterative repair-like interaction patterns, whereas CrewAI tends to be smoother for structured pipelines.

3.4.4 RQ3: Prompt Engineering and Context Construction for Repair

Structured prompts as a first-order determinant

A consistent conclusion across the corpus is that prompt structure and context construction materially affect outcomes. The literature increasingly treats the prompt not as a single text string but as an **engineered interface** that encodes role, constraints, context, and output contracts.

Ding et al. [38] synthesise reasoning and planning strategies (Chain-of-Thought, Tree-of-Thought, Program-of-Thought) and argue that explicit decomposition reduces functional regressions in complex repair settings. **Singh et al.** [37] identify recurring agentic workflow primitives—planning, tool use, reflection, and multi-agent collaboration—with **reflection/self-critique** repeatedly associated with error reduction in code-generation tasks.

Pu et al. [53] examine proactive versus reactive agent behaviours in programming workflows, highlighting a design constraint relevant to automation: overly proactive behaviour can disrupt developer flow, suggesting that autonomous remediation systems should provide transparent intermediate artefacts and controllable intervention points.

Zamfirescu et al. [54] show that prompting for **multiple alternatives** and explicit trade-off discussion can improve solution quality, an idea transferable to accessibility repair where multiple fix patterns may satisfy a rule but differ in semantics, UI impact, or maintainability.

Retrieval, grounding, and context selection

Agentic repair systems fail frequently due to missing or misleading context. Accordingly, the corpus heavily emphasises retrieval and grounding strategies.

Ramirez and Ekler [55] link code complexity metrics (e.g., cyclomatic complexity, maintainability) to repair difficulty, motivating the inclusion of **complexity-aware metadata** in the prompt as a guide for decomposition or conservative edits. The broader principle,that grounding the model in code-structural context reduces hallucination and improves correctness,is corroborated across multiple systems in the corpus [32, 44].

Hybrid approaches appear when compliance with formal constraints matters. **Abiakl et al.** [56] explore neuro-symbolic workflows in which symbolic representations complement LLM generation for structured code tasks. Although their primary focus is code comment classification rather than repair, the underlying paradigm is conceptually transferable: WCAG conformance includes rule-like structure that can be partially encoded symbolically, providing a principled constraint layer over LLM-generated repairs.

Domain-specific prompting and debugging protocols

The corpus includes multiple studies where success depends on embedding domain-specific structures into the prompt.

Empirical evidence across the corpus consistently supports the use of exemplar-driven prompt templates for structured code transformations [37, 38]: few-shot demonstrations reduce ambiguity in output format and help the model infer the expected repair style without exhaustive specification. **Grotov et al.** [57] show that enforcing an explicit diagnose–hypothesise–fix–validate loop increases resolution rates for errors in computational notebooks, which is best interpreted as evidence for **process-shaped prompting**: prompts that impose a repair protocol, not just a target output.

3.4.5 RQ4: Validation Oracles for Automatically Generated Repairs

A layered oracle model

Repair validation is repeatedly identified as a central open problem, and the strongest systems adopt **multiple complementary oracles**. Across the included studies, five oracle classes recur:

(1) Syntactic validation. Checks that produced artefacts parse/compile. This is necessary but weak as a discriminator, especially for strong LLMs [31, 44].

(2) Functional validation via tests. Execution of pre-existing or generated tests is the most common oracle. **SWT-Bench** [58] argues that high-quality repair evaluation often requires **repair-targeted test generation**, because existing suites

may under-specify the bug. The broader implication is that validation quality is bounded by oracle completeness.

(3) Static analysis validation. Linters, security scanners, and compliance tools act as scalable secondary oracles. **VVF-AI** [59] exemplifies a pattern where the agent not only generates fixes but also actively verifies vulnerability claims, including proof-of-concept style validation. **AutoVerus** [60] represents the most rigorous endpoint of this axis by integrating formal verification for Rust proofs.

(4) Semantic/domain validation. Several systems introduce checks intended to preserve behavioural intent and avoid "surface fixes". **ASPIRE** [44] includes a semantic evaluator agent for behavioural equivalence, a capability that maps directly to UI/accessibility contexts where passing a rule check does not imply preserving component semantics.

(5) Multi-agent validation via debate/consensus. Independent validation agents reduce correlated failure modes. **Lin et al.** [49] report that Debate–Verify–Debug reduces bug rates relative to a single-agent generator, suggesting a practical path to improve precision when traditional oracles are incomplete.

Residual failure modes after automated validation

Even with layered oracles, residual failures are common. Synthesising evidence across the included studies, the most frequently reported failure modes are: API hallucination, partial regressions, superficial fixes that satisfy tests without resolving semantics, and violations of framework conventions [31, 46, 47]. These modes motivate explicitly **workflow-level safeguards**: iteration with feedback, constrained editing, and independent verification.

3.4.6 RQ5: Limitations and Research Gaps

The corpus reveals multiple gaps that are directly relevant to the dissertation problem setting.

Gap 1: Web accessibility (WCAG) is under-represented. Among the included agentic repair and transformation studies, none focus explicitly on WCAG-guided remediation as the primary repair domain. Existing work is dominated by functional bugs, compilation issues, security vulnerabilities, and general code generation. This absence defines a concrete research opportunity.

Gap 2: Weak coupling between detection outputs and repair prompts. Most systems assume that a structured defect specification is available. Few studies treat the transformation from tool reports (static analysis, scanners, CI logs) into high-quality repair prompts as a first-class problem. This "detection-to-repair" gap is central for accessibility, where violation reports are noisy, tool-dependent, and

often lack semantic intent.

Gap 3: Functional preservation for UI-centric repairs is poorly measured. Mainstream benchmarks (e.g., SWE-bench-like pipelines, HumanEval-style tasks, COMMIT0) emphasise unit-test-defined correctness. For UI and accessibility, correctness includes visual/interaction preservation and assistive-technology semantics, which are not captured by standard oracles.

Gap 4: Lack of transferable prompt engineering frameworks for domain-rule repair. Prompt engineering evidence is often context-specific. There is limited systematisation for domains governed by formal or quasi-formal standards (WCAG, CWE, MISRA), especially for transforming rule violations into constrained, maintainable code edits.

Gap 5: Limited cost/benefit and CI/CD deployment analysis. While general discussions exist about integrating GenAI into software development pipelines [61, 62], the corpus contains relatively few detailed evaluations that jointly quantify repair quality, latency, token/cost budgets, and operational constraints required for CI/CD-grade deployment.

3.5 Implications for the *a11y-autofix* Framework

The review findings inform the *a11y-autofix* design along five interconnected dimensions. The dominant evidence supporting hierarchical orchestration and role-based specialisation for complex software engineering tasks [30, 48], reinforced by studies on hardware-oriented multi-agent code generation [63] illustrating that the planner–executor pattern generalises across domains, motivates adopting an orchestrator that delegates to specialised agents for detection normalisation, context construction, repair generation, and validation.

The consistent evidence in favour of prompts that combine role definition, precise problem statements, code context, explicit constraints, exemplars, and output-format contracts [37, 38, 57] motivates the six-component prompt template described in Chapter 4, which instantiates this pattern for accessibility repair. Since no single oracle captures all relevant failure modes, the review further supports a layered validation pipeline combining syntactic checks, functional preservation, domain compliance, and quality constraints [44, 58, 60], directly motivating the four-layer validation design of *a11y-autofix*.

The pattern of high-performing agents operating iteratively and using validation results as feedback for subsequent attempts [31, 32] motivates bounded retries and controlled backoff strategies that balance quality improvements against computational cost. The consistent finding that grounding the model in code-structural context reduces hallucination and improves correctness [32, 55] supports embedding

WCAG snippets, framework conventions (React/JSX), and repair-pattern exemplars directly into the constructed prompt context rather than relying purely on implicit model knowledge.

Finally, the corpus also models how such a design ought to be *evaluated*. Ablation (attributing a system's performance to its parts by switching each one off in turn) is standard practice among the strongest included systems rather than an optional extra: AutoCodeRover, RepairAgent, AutoVerus, DS-Agent, MAGE, and SWT-Bench [31, 32, 58, 60, 63, 64] each isolate a key component this way: the fault-localisation and code-search steps, the search tools and finite-state controller, a proof-refinement phase, the case-retrieval and case-based-reasoning modules, the agent-role partitioning, and the test-sampling budget, respectively. That precedent directly motivates the two ablation studies this dissertation runs (Chapter 5): one to justify the prompting strategy, and one to test whether each structural component of the pipeline earns its place.

3.6 Threats to Validity

Following common validity framing for software engineering research [41], the review has the following threats.

The search string may miss relevant work that uses alternative terminology (e.g., "LLM agent", "copilot", "code assistant") or that omits the term "agent" despite exhibiting agentic behaviour. This threat is partially mitigated by broad AI terms in the query and by screening that prioritises agentic properties in full-text assessment. Relatedly, this synthesis draws *only* on the 55 systematically retrieved studies, enumerated as **[S1]**–**[S55]** in Appendix B.5; works cited elsewhere in the dissertation that fall outside this set were obtained through complementary channels (unsystematic citation chasing, foundational background literature, and grey literature such as standards and tool documentation), which are catalogued and justified in Appendix B.6 to keep the boundary between systematic evidence and supporting literature explicit. No systematic snowballing pass was performed over the seed set.

The search was executed in a single database (Scopus); the ACM Digital Library and IEEE Xplore were not queried independently. Because Scopus indexes the major ACM and IEEE venues relevant to this topic, the practical impact of this limitation is expected to be low, but studies published in non-Scopus-indexed outlets may have been missed. Future work should replicate the search in complementary databases to bound this coverage bias.

No formal quality assessment of the primary studies was carried out. Included studies were screened for *relevance* against the eligibility criteria of Section 3.2, but they were not scored on methodological rigour, and no study was excluded

or down-weighted on the basis of its evidential strength. The rapid-review format accepts this concession, trading appraisal depth for timeliness [65], and it has a direct consequence for how the synthesis should be read. When a claim is reported here as a point of convergence, that reflects how frequently a mechanism *appears* in the corpus rather than how well it has been *demonstrated*: a preliminary result and a thoroughly validated one carry equal weight in the counts. Design decisions in Section 3.5 are accordingly justified as informed and non-arbitrary choices, with their merit settled by the experiments in Chapter 6 rather than by this chapter.

The synthesis rests on a single coder, with no second-researcher review, no independent double-coding, and consequently no inter-rater agreement statistic (Section 3.4.1). Category boundaries—most consequentially the boundary that produces the architectural patterns and the five gaps—are therefore one analyst's unchecked reading, and a different analyst could plausibly draw them elsewhere. Appendix B.5 publishes the per-study assignment so that a reader can audit this threat instead of having to take the acknowledgement on trust.

The synthesis reflects Scopus-indexed literature from 2020 through early 2026. Given that 2024–2025 dominate the corpus, the field is likely to evolve rapidly beyond the review cut-off.

Heterogeneity across tasks, benchmarks, and reporting conventions limits quantitative aggregation. The synthesis is therefore thematic with descriptive statistics, which is methodologically consistent with RR practice [65].

The topic is rapidly evolving and the review window, while deliberately bounded, may not capture the most recent preprints or workshop papers that precede formal indexing. This threat is treated as residual and acknowledged as a limitation of any time-bounded evidence synthesis in a fast-moving field.

3.7 Summary and Conclusions

This Rapid Review screened 138 Scopus records and synthesised 55 included studies (2020–early 2026) on AI-agent-based code repair, transformation, and generation, yielding five interconnected conclusions that inform the core design choices of the *a11y-autofix* framework. The conclusions describe where the retrieved literature converges; whether the resulting choices are *effective* is a separate question, settled empirically in Chapter 6.

Agentic repair systems are rapidly maturing (RQ1): the dominant shift is from template- and search-based automated program repair to autonomous LLM agents with iterative tool-using workflows, and systems such as RepairAgent and AutoCodeRover demonstrate that autonomy and iteration are essential for multi-step bugs and realistic issue settings. Multi-agent designs converge on a small set of

robust patterns (RQ2): hierarchical planner–executor architectures, role-based specialisation, and feedback-driven pipelines recur across systems, supported by practical orchestration frameworks such as LangGraph/LangChain, AutoGen, and CrewAI. Prompt and context engineering are first-order drivers of repair quality (RQ3): structured prompts with explicit constraints and output contracts, combined with retrieval-based grounding, reduce hallucination and improve reliability across the included studies. Validation requires composite oracles (RQ4): layered validation combining tests, static analysis, semantic checks, and independent verification agents is repeatedly associated with stronger outcomes, and no single oracle is sufficient. Finally, five key gaps motivate this dissertation (RQ5): web accessibility is underrepresented as a repair domain; the transformation from detection reports to repair prompts is weakly addressed as a first-class problem; UI-centric functional preservation is poorly measured by standard benchmarks; systematic prompt frameworks for rule-governed domains are scarce; and CI/CD cost–benefit analyses remain thin.

These five conclusions are the specification from which *a11y-autofix* is derived, as mapped in Section 3.5. The next chapter describes how the framework was built.

Chapter 4

The *a11y-autofix* Framework

This chapter presents the engineering contribution of this dissertation: the *a11y-autofix* framework, a pipeline that carries a React/TypeScript source file from accessibility violation to validated patch without human intervention. The chapter is descriptive of the artefact. How the artefact was evaluated — the research paradigm, the benchmark corpus, the experimental protocol, and the threats bounding the results — is the subject of Chapter 5.

Section 4.1 sets out the four-phase architecture and the contract between phases. Sections 4.2 to 4.5 then walk through each phase in the order a file experiences it: detection by four concurrent scanners, assembly of the six-component prompt, adaptive routing to a repair agent, and the four-layer validation gate that a patch must clear before it is applied. Section 4.6 covers the implementation decisions that make the framework reusable and locally executable, and Section 4.7 traces a single real violation end to end so that the abstractions of the preceding sections can be seen operating on concrete code.

4.1 Pipeline Overview

a11y-autofix is an automated repair framework that processes a React/TypeScript source file through four sequential phases: detection, prompt engineering, repair, and validation. Figure 4.1 illustrates this progression. Each phase has a well-defined input and output contract, and failure at any point produces a structured diagnostic rather than a silent error. The following sections walk through the pipeline in the order a file experiences it.

Figure 4.1 shows the phases as a left-to-right progression, which conveys the order of the contracts but not the control flow: it hides the decision points at which a file leaves the pipeline early, and it flattens the retry loop into a single arc. Because those decisions determine what actually happens to any given file, Figure 4.2 restates the same pipeline as an activity diagram, with each branch made explicit. The two

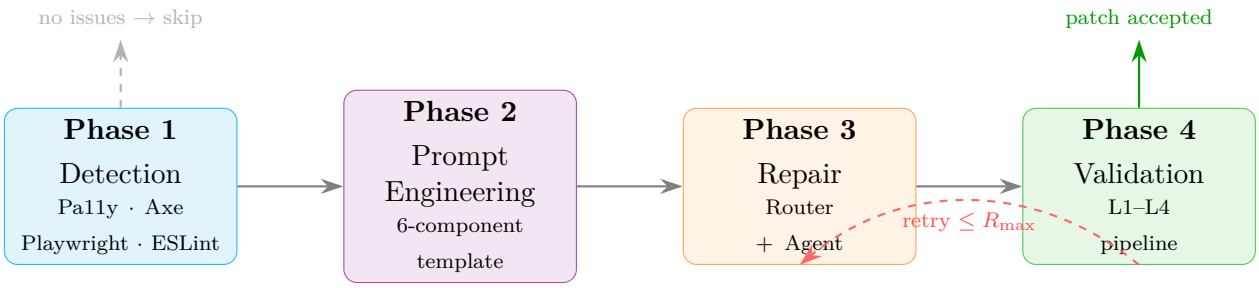


Figure 4.1: The four-phase *a11y-autofix* pipeline for a single source file. A dashed red arc represents the bounded retry loop: when a patch fails validation, structured feedback is passed back to the repair phase for up to $R_{\max} = 3$ attempts before the file is recorded as a repair failure.

figures describe one system: the first is the architecture, the second the behaviour.

4.2 Phase 1: Detection

The first thing the framework does with a source file is scan it for accessibility violations. Rather than relying on a single tool, the framework runs four scanners whose mechanisms span both runtime and compile-time analysis: **Pa11y** [3], **Axe-core** [4], **Playwright+Axe** [22], and **ESLint with jsx-a11y** [23].

The three runtime scanners are co-ordinated through an HTML harness: the framework wraps the React component source in a minimal test page, serves it from a local HTTP server to avoid `file://` protocol restrictions, and launches the browser-based tools against that URL. Pa11y drives headless Chromium through HTML_CodeSniffer, reporting violations as WCAG guideline codes with CSS selectors into the rendered DOM. Axe-core evaluates the live DOM with its zero-false-positive engine, enriching each finding with a structured impact level and a stable rule identifier. Playwright launches a full browser instance and waits for all mount effects and asynchronous state updates to settle before injecting axe-core, capturing violations that only materialise after React's reconciliation completes. ESLint operates differently: it is invoked directly on the TypeScript source file without any rendering step, analysing the JSX abstract syntax tree with 21 rules from the `eslint-plugin-jsx-a11y` plugin. Because it works at the source level, ESLint produces findings whose selectors are source coordinates (`filename:line:column`) rather than DOM paths, making its output directly usable for patch generation without an additional selector-to-source mapping step.

The four tools are invoked concurrently using Python's `asyncio` event loop, and their raw outputs are merged through a deduplication step that uses a content-addressed key composed of the normalised selector (or source coordinate) and the WCAG criterion identifier. Findings that survive deduplication are assigned a con-

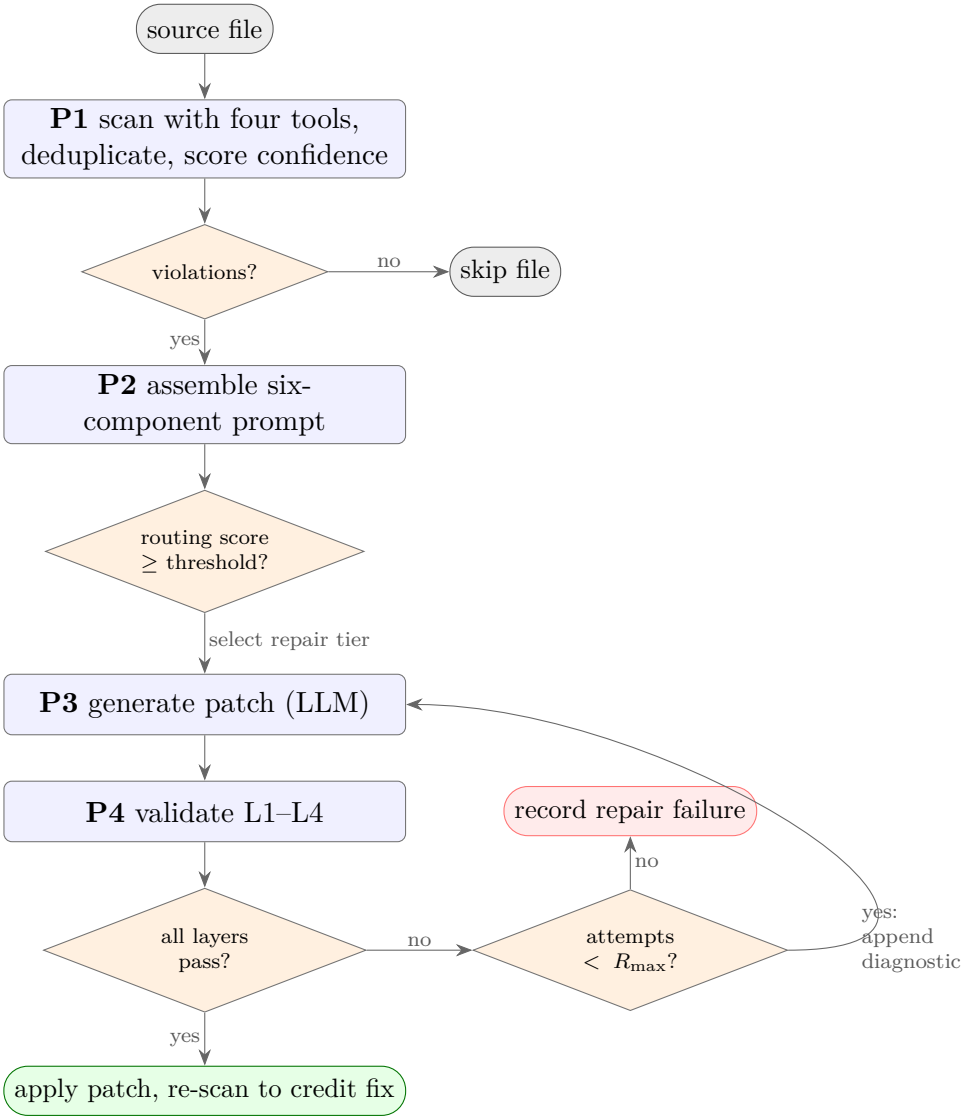


Figure 4.2: Activity view of the same pipeline. Three decisions govern what happens to a file: whether any violation was detected at all, which repair tier the routing score selects, and whether the generated patch clears every validation layer. A rejected patch returns to the repair activity with the failing layer's diagnostic appended to the prompt, up to $R_{\max} = 3$ attempts; only a patch that clears all four layers is applied and only then is a fix credited, by re-scan. The 1,216 violation-free files of the corpus exit at the first decision.

fidence level: HIGH when two or more independent scanners agree, MEDIUM when a single scanner reports at critical or serious severity (exploiting axe-core's zero-false-positive guarantee at those levels), and LOW otherwise. All detected violations are passed to the repair pipeline regardless of confidence level; the confidence level influences the priority order in which issues are presented to the LLM (high-confidence issues appear first in the prompt), not the filtering decision. LOW-confidence findings represent weak or ambiguous scanner signals (single-tool detections at minor severity), and in the canonical experiment achieve a zero fix rate across all three models and all three repetitions (Section 6.2). Each violation is then classified into one of seven issue types and sorted by confidence and impact so that the most consequential violations appear first when context window constraints require truncation.

4.3 Phase 2: Prompt Engineering

A violation report from a scanner is structurally thin: it contains a violation code, a CSS selector, and a terse message. That is enough to tell a developer what is wrong, but not enough for an LLM to generate a correct, minimal, framework-appropriate patch. The prompt engineering phase bridges this gap by assembling a structured prompt from six components, each targeting a specific failure mode identified in preliminary experiments (Section 2.5).

The six components are assembled in sequence. A **role definition** establishes the model as a React accessibility expert, activating relevant knowledge patterns from pre-training. A **violation context** block expands the raw scanner output into a full description of the WCAG criterion, its rationale, and its real-world impact on users, giving the model the semantic intent that the scanner report omits. A **code context** block supplies the complete source file, its TypeScript prop interface, and its event handlers, so the model can reason about what changes are architecturally safe. **Explicit constraints** enumerate what the model must not do: alter prop interfaces, introduce non-existent APIs, add unnecessary code. **Few-shot exemplars** demonstrate the expected repair pattern through violation–fix pairs retrieved by cosine similarity from a curated library, ensuring that examples are relevant to the current instance rather than generically representative. The exemplar library is organised by violation category and contains hand-authored pairs covering the principal WCAG violation types present in the benchmark: ARIA attribute additions and corrections (covering `aria-label`, `aria-hidden`, `role` assignments), keyboard handler additions (pairing `onClick` with `onKeyDown`), form label associations (`htmlFor` and `aria-labelledby` patterns), and alt-text assignments for image elements. The retrieval mechanism in `PromptBuilder` always includes a keyboard exemplar (present in every repair context) and adds ARIA, label, or semantic ex-

emplars adaptively when the current violation belongs to those categories, selecting the pair whose violation description has the highest cosine similarity to the current instance. The complete exemplar set is included in the open-source replication package. Finally, an **output format specification** defines the exact response schema, enabling deterministic parsing and validation downstream.

The absence of any component is not a neutral omission. During framework development, qualitative ablation observations were recorded by constructing reduced prompt variants, each omitting exactly one component, and inspecting the LLM outputs on a convenience set of approximately 30 violations spanning the four most prevalent issue types (ARIA attribute, keyboard, label, and alt-text). The observations across these trials are as follows. Removing Component 2 (violation context) consistently produced superficial or semantically incorrect repairs: without the full WCAG normative description and the user-impact statement, the model generated `alt="decorative image"` for decorative images rather than the required `alt=""`, and supplied generic `aria-label="button"` values that fail to convey the element's specific purpose. Removing Component 4 (explicit constraints) increased over-engineering and API hallucination: the model introduced custom React hooks, unnecessary `useState` calls, and non-existent ARIA attributes (`aria-role="navigation-button"`) for issues requiring only a single attribute addition. Removing Component 5 (few-shot exemplars) preserved semantic correctness in many cases but degraded consistency of repair style, producing fixes structurally inconsistent with the codebase's conventions (e.g., using inline `style` where the project's convention is CSS Modules). These observations are qualitative and not the product of a controlled, statistically powered experiment; they are reported here to make transparent the empirical basis for the template's six-component design. The quantitative counterpart to these observations is the architectural ablation that answers RQ1 (Section 5.4.3), which switches off the framework's structural components, including the few-shot exemplars discussed here, one at a time and measures the effect on the fix rate; as Section 6.10 reports, that controlled test does not reproduce the qualitative intuition that each component lifts performance, a tension taken up in the discussion.

4.4 Phase 3: Adaptive Routing and Repair

Not all violations benefit from the same repair strategy. Adding a missing `alt` attribute to an image is a localised, single-attribute insertion that requires no understanding of the broader codebase. Fixing a colour contrast violation may require tracing CSS variable definitions through a theme configuration file that lives in a different directory. Routing every violation through the most capable and most ex-

pensive agent would be wasteful; routing complex violations through the simplest agent would produce incomplete patches.

The framework resolves this by computing an integer routing score for each file based on violation characteristics: the presence of `contrast` or `semantic` violations, the total issue count, the number of distinct issue types, and the maximum complexity level. Files scoring below a calibrated threshold are routed to the **SWEAgent**, a custom-built agent implemented specifically for this framework, which handles surgical, line-targeted edits. Files scoring at or above the threshold are routed to **OpenHands** [16], a sandboxed coding environment with shell and file system access that supports multi-file reasoning. When the primary agent (SWEAgent) exhausts its bounded retry budget without producing a valid patch, the file is recorded as a repair failure; no further escalation agent is invoked.

The SWEAgent is the primary repair engine and executes through two tiers, selected at runtime depending on what tooling is available in the execution environment.

1. **CLI mode** (*SWE-agent CLI present*). The agent invokes the SWE-agent command-line interface [15] as a subprocess, passing the task description and the target file. The CLI receives the prompt assembled in Phase 2, calls the local Ollama endpoint, and drives its agent–computer interaction loop autonomously.

2. **Direct surgical mode** (*CLI absent* — implemented as `DirectLLMAgent`). The agent submits the same prompt directly to the LLM with an instruction schema that requests `FIND/REPLACE` patch blocks rather than a full file rewrite. Each block specifies an exact string to locate in the original file and the replacement string, minimising the diff surface and reducing the risk of unintended changes to surrounding code. The agent applies each block sequentially to the original source and passes the result to the validation pipeline.

   Inner fallback: if the patched file is not syntactically well-formed (checked via Babel parsing), the agent abandons the patch blocks and instead interprets the full file content from the first `tsx` code block in the response.

Important: in the execution environment used for this dissertation, the SWE-agent CLI was not installed; *all repairs in the canonical experiment and the architectural ablation were therefore performed by the* ***`DirectLLMAgent`*** (tier 2 above). The CLI mode (tier 1) was not exercised in any reported experiment. Claims about "SWE-agent" repairs in this dissertation refer to the SWE-agent-style FIND/REPLACE prompt format delivered via `DirectLLMAgent`, not to the full SWE-agent autonomous agent loop. This limitation is also noted in the appendix (Appendix D).

This design allows the framework to remain fully functional in environments where only a local Ollama instance is present.

Once the agent is selected, it receives the assembled prompt and returns a patch. If the patch fails validation, the structured failure diagnostic is appended to the prompt and the agent retries, up to $R_{\max} = 3$ attempts. After exhaustion, the file is recorded as a repair failure and the pipeline moves to the next file.

4.5 Phase 4: Validation

Every generated patch must pass four sequential quality gates before it is written to disk. The gates are ordered from cheapest to most expensive, so that obviously invalid patches are rejected early without incurring the cost of full TypeScript compilation.

A note on "validation". The component is named the *validation pipeline* throughout this dissertation and in the implementation, but the name is inherited from the software rather than chosen against the standard vocabulary, and the mismatch should be stated plainly. Under the IEEE and ISO/IEC/IEEE 12207 distinction, *verification* asks whether an artefact conforms to its specification—are we building the product right—while *validation* asks whether the artefact meets the user's actual need—are we building the right product. By that definition, all four layers perform *verification*. Layer 1 checks conformance to an output format, Layer 2 to the component's declared interface, Layer 3 to the WCAG rule that flagged the violation, and Layer 4 to quality thresholds. Each compares the patch against a specification.

Validation proper—confirming that a repaired component actually serves a user of assistive technology better than the original—is *not* performed anywhere in this work. No screen-reader session, no keyboard-navigation trial, and no user study was conducted, and no human ever judged a patch (Section 5.7). This is not a quibble about vocabulary: the gap between the two words is the same gap that produces the semantic ceiling of Chapter 7, and the reason the reported fix rate is described as *scanner-verified repair* rather than as accessibility improvement. The name is kept for consistency with the released code; the reader should read it as "the verification pipeline" throughout.

Layer 1 (Syntactic) verifies that the LLM output contains exactly one well-formed code block in the expected format, catching the most common output formatting failure: models that produce partial files, multiple code blocks, or prose mixed with code. Layer 2 (Structural Interface Preservation) then extracts the component's TypeScript prop interface, export signatures, and event handler bind-

ings from both the original and the patched file using AST analysis, and verifies they are identical. This is the most consequential gate, since a patch that removes a prop or renames an event handler will break every component that consumes this one, silently and at runtime. Layer 3 (Domain Compliance) applies lightweight heuristic checks specific to each violation type: for an `alt-text` violation it verifies that the patched file contains an `alt` attribute on the affected element, and for an `aria` violation it verifies that the required ARIA attribute is present. These checks do not re-run the full scanner suite (which would multiply validation time by 3–5×); they provide a fast proxy signal that the model actually addressed the violation rather than returning the original file unchanged. Layer 4 (Code Quality) runs the TypeScript compiler in `-noEmit` mode, measures the cyclomatic complexity delta, and checks the ratio of added to removed lines; a patch that introduces type errors, substantially increases complexity, or adds ten times more code than it removes is flagged as over-engineered and rejected.

A patch that clears all four layers is written to disk, but passing validation is not the same as resolving the violation. The two are kept deliberately separate. Validation asks whether the patch is *safe and plausible*; resolution asks whether the violation has actually *disappeared*. To answer the second question without circular reasoning, the framework re-runs the full browser-based scanner suite against the patched file and credits a violation as fixed only when the corresponding finding no longer appears in the re-scan. Crucially, the credit is bounded to the *net* reduction per WCAG criterion: if a patch reshapes the DOM so that an old finding's selector changes but an equivalent violation persists under a new selector, the criterion's count does not fall and no credit is awarded. This re-scan oracle, rather than the fast Layer 3 proxy, is what populates the issue-fix-rate numerator reported in Chapter 6; the proxy exists only to fail cheap patches early, while the oracle is the authoritative measure of repair.

Figure 4.3 summarises the four layers and their primary targets.

4.6 Implementation: Modularity, Reuse, and Local Execution

The four phases are implemented as a single installable Python package (`a11y-autofix`) rather than a monolithic script: detection, routing, repair, and validation each live in their own module, and the detection tools sit behind one common adapter interface, so a scanner can be added or swapped without touching the rest of the pipeline. That separation paid off directly in the experiments. The ablation harnesses of Section 5.4 do not re-implement the pipeline; they import

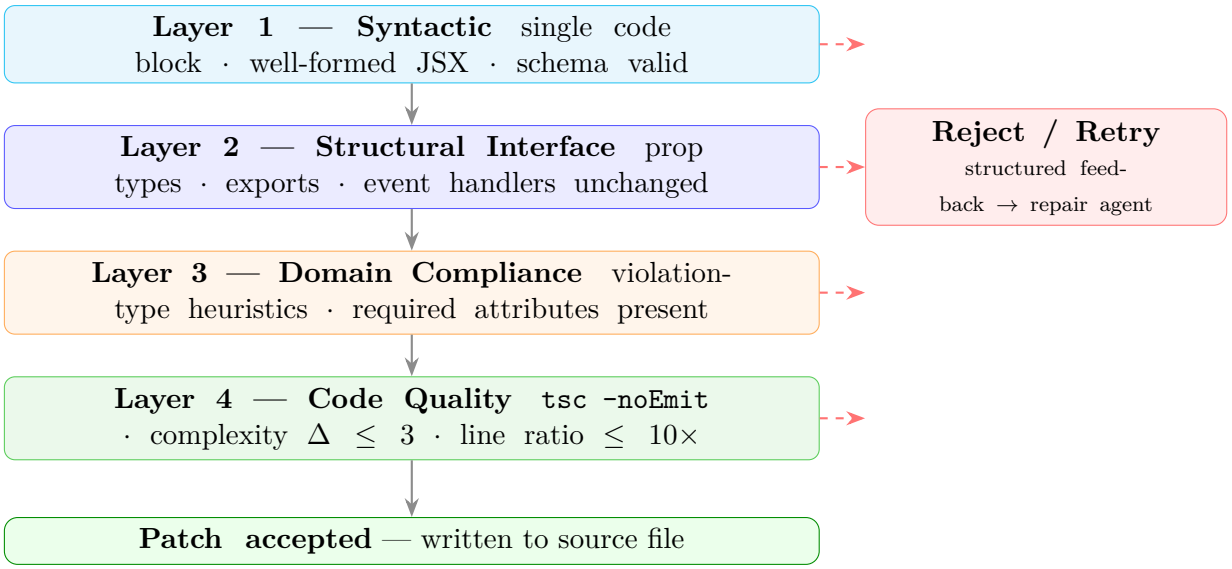


Figure 4.3: The four-layer validation pipeline. Each layer applies progressively deeper checks; rejection at any layer feeds structured failure diagnostics back to the repair agent for a bounded retry.

and run the very same production components (the validation pipeline, the prompt builder, the agent interface, and the shared configuration types), so each ablation measures the real system rather than a parallel copy of it. Reuse serves a correctness purpose here in the sense intended by software-reuse practice: the artefact under evaluation is the artefact that would be deployed.

Building and running this locally surfaced concrete engineering problems worth recording. The concurrency that Phase 1 relies on is bounded by a semaphore, so a commodity machine launching several headless browsers at once is not overwhelmed; even so, browser-based scanning proved the fragile part, and each component had to be served over a local HTTP harness rather than a `file://` URL to stop Pa11y and Playwright from timing out on load. Because a full local run takes hours, the experiment runner checkpoints after every condition and skips any cell whose results already exist, so an interrupted run resumes where it left off instead of restarting, a small piece of engineering that is what makes long, unattended runs on a single machine practical at all.

That single machine is the point. The whole loop (detection, model inference, the four validation layers, and the browser re-scan that credits each fix) runs end to end on one consumer laptop (Section 5.4.2), with the language models served locally through Ollama and nothing sent to a cloud API. This was a deliberate constraint rather than an accident of budget: if the aim is accessibility repair that a developer can actually adopt, the proof of concept has to be that it runs on the kind of machine that developer already owns. The canonical experiment is exactly that demonstration: the entire study was produced on a laptop, offline, at no per-token

cost.

4.7 Worked Example: A Single Violation, End to End

With the pipeline and its implementation now described, the four phases are easier to trust once seen working on a real case. This section follows a single violation, drawn verbatim from the canonical experiment, all the way from the scanner's terse report to a corrected, re-scan-verified component. The example is deliberately small, so that the reader can hold the entire file in mind; the same machinery applies unchanged to larger components.

The component is `index.tsx`, a header title from the open-source *Suika* vector editor. In full, it is just this:

```
import './style.scss';
import { GithubOutlined } from '@suika/icons';
import { type FC } from 'react';

interface IProps {
  value: string;
}

const Title: FC<IProps> = ({ value }) => {
  return (
    <div className="suika-header-title">
      <a href="https://github.com/F-star/suika" target="_blank">
        {value}
      </a>
    </div>
  );
};

export default Title;
```

Phase 1: the finding, and why it is hard to act on. The detection layer returns a single line:

```
[LABEL] WCAG 2.4.4 - Ensure links have discernible text | selector: ‘a‘
```

To an accessibility specialist this is enough. To a developer without accessibility experience, which describes the overwhelming majority of practitioners, it is nearly opaque, and instructively so. The report names a criterion by number (2.4.4) rather than by what it means; it points at a bare element type (`a`) rather than at a line; and, most misleading of all, it appears to be *wrong*. The link plainly contains `{value}`, so a reasonable first reaction is "but it already has text, this is a false alarm." The subtlety that resolves the contradiction is one the tool never states: `value` is a runtime prop, and the scanner, rendering the component in isolation, finds the link with no discernible name because nothing guarantees `value` is non-empty, meaningful text. Acting correctly here requires knowing what WCAG 2.4.4 demands, why a dynamic prop is not a reliable accessible name, and which of several possible remedies (visible text, an `aria-label`, an `aria-labelledby`) is appropriate. That is exactly the chain of contextual judgement the prompt-engineering phase exists to supply.

Phase 2: the assembled prompt. The framework expands the one-line finding into a structured prompt. The system message fixes the role, the editing discipline, and the exact output contract:

```
You are a surgical accessibility fixer for React/TypeScript code.
RULES:
- MINIMAL CHANGES ONLY - change the absolute minimum to fix each issue
- Each patch must be a precise find-and-replace
- Preserve all whitespace, indentation, and formatting around changed lines
OUTPUT (preferred for <=5 changes):
PATCH 1:
FIND: ‘<exact original line>‘
REPLACE: ‘<corrected line>‘
```

The user message then supplies the component in full, the conformance target, and the enriched violation, so the model reasons over the same context a developer would have open:

```
Fix accessibility issues in: index.tsx
WCAG: WCAG2AA

‘‘‘tsx
... (the full component shown above) ...
‘‘‘

Issues to fix:
1. [LABEL] WCAG 2.4.4 - Ensure links have discernible text | selector: ‘a‘
```

Phase 3: the repair. Routed through the surgical agent, the model returns a single find-and-replace patch, the minimal edit the system prompt asked for:

```
PATCH 1:
FIND: ‘<a href="https://github.com/F-star/suika" target="_blank">‘
REPLACE: ‘<a href="https://github.com/F-star/suika" target="_blank"
         aria-label="GitHub repository">‘
```

The choice is the right one. Rather than touch the dynamic `{value}`, the model adds a stable `aria-label` that gives the link a discernible name regardless of what `value` holds at runtime; a screen reader now announces “GitHub repository, link” instead of an unnamed control.

Phase 4: validation and verification. The patch clears all four validation layers: it is one well-formed code block (L1), it changes no prop, export, or handler (L2), the targeted element now carries an accessible name (L3), and it compiles with a negligible complexity delta (L4). The framework then re-runs the browser scanner against the patched file, and the WCAG 2.4.4 finding is gone: the fix is credited not because the model claimed success but because the violation no longer reproduces. In the canonical experiment this repair was produced on the first attempt, cost 55 output tokens and about ten seconds, and, reassuringly for a stochastic generator, was reproduced identically in all three repetitions. The corrected component differs from the original by exactly one attribute:

```
<a href="https://github.com/F-star/suika" target="_blank"
   aria-label="GitHub repository">
  {value}
</a>
```

This is the case the framework handles best: a structural, attribute-level repair with an unambiguous correction criterion. The contrast with the semantic violations that resist automation (Section 6.9) is precisely what the aggregate results make quantitative.

4.8 Summary

This chapter described the artefact. *a11y-autofix* carries a React/TypeScript file through four phases with an explicit contract between each: detection compiles the scanners’ output into deduplicated, classified violations; prompt engineering turns each violation into a six-component instruction that supplies the context a bare

report omits; adaptive routing selects a repair tier by complexity score; and the four-layer gate decides whether the generated patch is allowed to reach disk.

Two design commitments run through all four phases. The first is that nothing the model produces is trusted on its own: a patch is applied only after clearing syntax, interface preservation, domain compliance, and quality checks, and a rejection returns to the agent as structured feedback rather than as a discard, turning one-shot generation into a bounded search of at most $R_{\max} = 3$ attempts. The second is that the checks are *verification* against specifications, not validation against user need—a distinction stated in Section 4.5 and carried through the rest of the dissertation.

The worked example of Section 4.7 traced one real violation from scanner report to accepted patch, showing the abstractions operating on concrete code. What the chapter deliberately did not do is establish that any of this works: the framework is described here, and Chapter 5 sets out how it was measured and with what limits on the resulting claims.

Chapter 5

Evaluation Design

Chapter 4 described what was built. This chapter describes how it was evaluated, and with what limits on what the evaluation can support.

Section 5.1 states the research paradigm and the four research questions that govern the design. Section 5.2 states the five hypotheses a priori, so that the results reported in Chapter 6 can be read against commitments made before the data were collected rather than after. Section 5.3 documents the construction of the benchmark corpus, and Section 5.4 the three studies run against it. Section 5.5 defines every outcome measure and the oracle that credits a repair, Section 5.6 the statistical procedures applied to them, and Section 5.7 the threats that bound the interpretation of everything that follows.

5.1 Research Paradigm

This dissertation adopts a *Design Science Research* (DSR) paradigm [66], in which the primary output is a concrete artefact, the *a11y-autofix* framework, and knowledge is generated through its construction and evaluation in realistic problem settings [67]. The work pursues two objectives in parallel: an engineering objective (automating WCAG violation repair in React/TypeScript codebases) and an experimental objective (characterising which design choices are associated with better outcomes and under what conditions).

The evaluation oracle is composed of automated accessibility scanners, which algorithmically verify roughly 30–50% of WCAG criteria. A patch that satisfies the scanner may still violate criteria the scanner does not test. All quantitative results in this dissertation should therefore be interpreted as scanner-level performance indicators rather than absolute accessibility correctness scores. This limitation is revisited in Section 5.7.

The DSR cycle manifests across four chapters. The *problem* is the detection–correction gap in WCAG accessibility repair (Chapter 2). The *artefact* is the *a11y-*

autofix framework, specifically the six-component prompt template, the adaptive routing mechanism, and the four-layer validation pipeline (Chapter 4). The *evaluation* is designed in this chapter and has three arms, whose results appear in Chapter 6: a prompting ablation that isolates the effect of the prompting strategy on a single model (RQ2); the canonical experiment that compares three open-weight LLM architectures across 168 confirmed violations (RQ3 and RQ4); and an architectural ablation that switches off each structural component in turn to measure its contribution (RQ1). The *contribution to knowledge* is a failure mode taxonomy, a reproducible benchmark corpus, and the empirical characterisation of the automation ceiling (Chapter 7). Mapping these to the Hevner et al. design science guidelines [66]: the artefact is a design *method* (the prompt template) and a design *process* (the four-phase pipeline); the evaluation uses a controlled experiment in a realistic codebase context; and the knowledge contribution includes both the taxonomy (a *construct* for understanding LLM repair failures) and the benchmark (a *model* of the repair task's difficulty distribution).

5.1.1 Research Questions

The four research questions that govern the experimental design were introduced in Section 1.3, where each is accompanied by the rationale for asking it. They are restated here verbatim, without that rationale, because everything in the remainder of this chapter—the hypotheses, the corpus, the three studies, and the outcome measures—is designed against their exact wording, and a reader working through the design should not have to hold the phrasing in memory from ninety pages earlier. Together they cover the framework's internal structure (RQ1), its configuration sensitivity (RQ2), its behaviour across model choices (RQ3), and the unevenness of its performance across violation types (RQ4):

RQ1 Does a structured six-component prompt template yield significantly higher repair success rates than ablated versions of itself?

RQ2 Do prompting strategies (zero-shot, few-shot, and chain-of-thought) produce measurable differences in repair success rate and token efficiency?

RQ3 Do distinct locally deployable LLM architectures differ significantly in repair success rate, mean time to repair, and token cost?

RQ4 Is repair success rate associated with violation complexity category, and if so, which categories are most and least amenable to automated repair?

5.2 Formal Hypotheses

The following null and alternative hypotheses are stated a priori to make the experimental evaluation falsifiable and to mitigate p-hacking risk. All tests are conducted at $\alpha = 0.05$. H1–H4 are comparative hypotheses about observed behaviour within the evaluated system configuration and are grounded in a scanner-based oracle; they should be understood as heuristic-comparative rather than strictly confirmatory claims.

5.2.1 H1: Effect of Prompt Template Completeness (RQ1)

H_0^1 For each reduced ablation condition c_j ($j = 1, \ldots, 6$), the observed SR under c_j is not significantly lower than the SR of the full six-component template, as assessed by McNemar's test at $\alpha = 0.05$ on paired binary outcomes over the same benchmark files.

H_1^1 At least one reduced condition c_j yields a significantly lower SR than the full template (McNemar's $p < 0.05$, Cliff's $|\delta| \geq 0.147$), indicating that the absent component contributes a practically non-negligible increment to scanner-level repair success.

Test McNemar's test [68] on paired binary repair outcomes (same file, full vs. ablated condition).

Effect size Cliff's δ [69]; minimum threshold $|\delta| \geq 0.147$ (small effect).

Note SR here refers to scanner-level correctness only; it does not imply WCAG conformance as assessed by human experts.

Outcome This hypothesis was operationalised by the architectural ablation (Section 6.10), with two deviations from the plan stated a priori stated here for transparency: it ablates the framework's structural components (reflection loop, few-shot exemplars, WCAG semantic context, internal validation) rather than the six template slots one at a time, and it tests per-file fix rates with the Wilcoxon signed-rank test under Holm–Bonferroni correction rather than McNemar on binary file outcomes, the file being the correct independent unit. No ablated condition is significantly worse than the full pipeline, so H_0^1 is not rejected; with sixty of sixty-six files tying, the test is underpowered and the null is read as "no detectable contribution," not proven irrelevance.

5.2.2 H2: Effect of Prompting Strategy (RQ2)

H_0^2 Repair effectiveness (IFR) does not differ across zero-shot, few-shot, and chain-of-thought prompting strategies when evaluated on the same benchmark files with the same fixed LLM.

H_1^2 At least one strategy differs from the others in IFR (two-sided; Kruskal-Wallis omnibus $p < 0.05$; post-hoc pairwise Mann-Whitney U with Bonferroni correction; Cliff's $|\delta| \geq 0.147$). The hypothesis is deliberately non-directional: the comparison was designed to discover which strategy is superior, not to presume that the more elaborate strategies (few-shot, chain-of-thought) would win.

Secondary criterion Beyond IFR, the preferred strategy is the one that also minimises token cost per resolved violation, so that the selection reflects both effectiveness and efficiency.

Test Kruskal-Wallis [70] omnibus test; post-hoc pairwise Mann-Whitney U [71] with Bonferroni correction; Friedman test for the paired, within-corpus design; effect sizes via Cliff's δ [69].

Outcome Confirmed (Section 6.4): zero-shot significantly outperforms both few-shot (Cliff's $|\delta|$=0.34, p=0.019) and chain-of-thought ($|\delta|$=0.47, p=0.0004); Kruskal–Wallis and Friedman both $p \approx 0.0003$.

5.2.3 H3: Practical Utility Threshold (RQ3)

H_0^3 The framework's aggregate IFR does not exceed the practical utility threshold of 0.70 on the benchmark corpus: IFR < 0.70.

H_1^3 The framework achieves IFR ≥ 0.70, indicating that the pipeline resolves a sufficient majority of scanner-detected violations to function as a useful first-pass automation layer in a developer workflow.

Threshold justification The 0.70 threshold was set a priori as an operational convention before data collection. A framework meeting it reduces the manual correction burden to less than one third of baseline, which we take as the minimum bar for practical utility. The figure is analogous to quality-gate targets commonly adopted in automated testing (e.g., branch coverage floors) and should be read as a practical convention, not a formally derived cutoff. See Section 6.5 for the full rationale.

Test Comparison of the cross-repetition mean IFR (Table 6.13) against the 0.70 threshold. Supplementary inter-model comparison: Kruskal–Wallis omnibus + pairwise Mann–Whitney U with Bonferroni correction + Cliff's δ on per-file IFRs (n=33 files; Section 6.11).

Secondary metrics MTTR (continuous) and token cost (continuous) are reported as descriptive secondary outcomes.

Scope Conclusions apply to the specific model versions and quantisation settings evaluated; they do not extend to cloud-hosted or instruction-tuned variants not included in the study.

Outcome H3 is rejected for all three models. Best observed cross-repetition mean: `qwen2.5-coder-7b` at 0.544 (95% hierarchical bootstrap CI [0.360, 0.753]), 16 pp below the threshold. Supplementary inter-model comparison: Kruskal–Wallis H=0.16, p=0.92; all pairwise $p_{\text{Bonferroni}} \approx 1.0$; all Cliff's $|\delta| \leq 0.08$. The three models are statistically indistinguishable on repair quality under this corpus.

5.2.4 H4: Differential Repair Success Across Issue Categories (RQ4)

H_0^4 The Issue Fix Rate (IFR) does not differ significantly across the seven accessibility issue categories, as assessed by the Kruskal-Wallis test at $\alpha = 0.05$.

H_1^4 IFR is associated with a statistically significant difference across at least two categories (Kruskal-Wallis $p < 0.05$; post-hoc pairwise Mann-Whitney U with Bonferroni correction), with categories classified as *simple* complexity (`alt-text`, `aria`, `label`) exhibiting higher observed IFR than categories classified as *complex* (`contrast`, `semantic`).

Note The simple/complex classification is an analytical construct specific to this study; different systems or LLMs could produce a different complexity ordering.

Minimum evidence Kruskal-Wallis significance plus at least one post-hoc pairwise comparison between a simple and complex category reaching Cliff's $|\delta| \geq 0.147$.

Outcome Descriptive only (Section 6.9): the inter-category KW test is not valid because two categories have $n \leq 7$ (alt-text: 7, keyboard: 1). Per-type IFR is reported in Table 6.11; the semantic-structure category (33% of corpus) drives the aggregate result at 14–35% IFR across models.

5.2.5 H5: Functional Regression Rate (RQ3, Descriptive)

Research question What fraction of accepted patches introduce new accessibility violations or structural regressions not present in the original file?

Operationalisation $\rho = |\{p : \text{new violations after } p\}| / |\{p : \text{accepted by pipeline}\}|$.

Nature **Descriptive**; no formal null hypothesis is stated a priori and no statistical test is applied. Results are reported as a point estimate with 95% confidence interval across all accepted patches.

Practical reference The dissertation uses $\rho < 5\%$ as a practical reference point, drawn by analogy with quality-gate ceilings in automated testing literature, to contextualise the observed values. This threshold was defined before data collection as an operational convention, not as a formal pre-specified null; the associated ✓/× notation in Chapter 6 reports whether the observed values fall below this reference, not whether a statistical hypothesis was confirmed or rejected.

5.3 Benchmark Dataset

A meaningful evaluation of the framework requires violations that come from real codebases, span the full range of violation categories and complexity levels, and are reproducibly pinned to a fixed code snapshot so that experiments can be replicated independently. Constructing such a dataset is a non-trivial research contribution in its own right; this section describes the full data collection pipeline from repository discovery to the final annotated corpus.

The collection process begins with a structured query to the GitHub Search API. Three fixed-point search terms are combined: the repository must be classified under the `TypeScript` primary language (ensuring that components can be parsed as TSX rather than plain JSX), must contain the string `react` in its topic list or description, and must satisfy a set of quality filters applied through the API's sort and filter parameters. The quality filters require at least 50 stars (a proxy for non-trivial use beyond a personal toy project), at least one commit within the preceding 12 months (ensuring the codebase is not abandoned), and a licence that is recognised as open source by the Open Source Initiative (necessary to permit redistribution in the replication package). Repositories matching these criteria are retrieved in pages of 100, sorted by star count to make the sampling reproducible given the deterministic ordering.

Repository discovery is followed by stratified sampling designed to prevent domain bias. A dataset composed exclusively of dashboard applications, for example, would over-represent ARIA violations typical of complex interactive widgets and under-represent the contrast and alternative-text issues more common in media-heavy sites. The stratification grid is $7 \times 3 \times 3$: seven application domains (e-commerce, government, healthcare, education, developer tools, dashboard, and social), three size tiers classified by component file count (small: fewer than 50 files; medium: 50–200 files; large: more than 200 files), and three popularity tiers based on star count. The target corpus was $N = 400$ repositories, with at least two repositories required per stratum to ensure representation even in rare combinations. The canonical experiment described in this dissertation was executed on a subset of 36 repositories (1,249 files): the portion of the target corpus that could be fully processed within the available computational budget. Evaluating the full 400-repository corpus across three models and three repetitions would require $400 \times 35 \times 9 \times 5.73/86{,}400 \approx 8.4$ days of continuous single-GPU time at the slowest model's throughput (CodeLlama, 5.73 s/file), and under a day at the mean throughput across the three models ($\bar{t} \approx 2.4$ s/file). The 36-repository corpus was selected to maximise domain diversity within the feasible execution window; the stratified sampling procedure ensures that all seven domains are represented. The domain classification is performed by a keyword and metadata classifier applied to the repository description, primary topics, and readme content.

For each candidate repository that passes the domain classifier, the framework clones the default branch and applies a set of structural filters. Monorepos without a clearly identifiable React application root, repositories composed primarily of auto-generated files (those containing the string `@generated` or produced by tools such as GraphQL codegen), and repositories where fewer than five TSX or JSX files contain component-defining `export` statements are excluded. After filtering, each accepted repository undergoes a snapshot operation: the framework records the full commit SHA-1 of the HEAD commit at collection time and computes SHA-256 digests for each component file. All subsequent scanning and experimentation operates on this frozen archive rather than the live repository, ensuring that the violations in the benchmark remain stable even if the upstream project's maintainers fix them between collection and evaluation.

The scanning phase applies the full four-tool detection pipeline described in Section 4.2 to every component file in the snapshot. All sampled files (up to 35 per project) are retained in the experimental corpus regardless of whether they contain violations. For each file, the scanner records either a structured violation manifest (file path, SHA-256 digest, WCAG criterion, issue type, confidence level, detecting tools, and raw findings per tool) or an empty result indicating no HIGH or MEDIUM

confidence violations. Files without confirmed violations are kept in the corpus and processed by the pipeline as regression guards: if any patch introduced a new accessibility violation into a previously clean file, it would be detected at oracle re-scan time. Only repositories that produce *no* violations at HIGH or MEDIUM confidence in any of their sampled files are excluded during the repository-selection phase, because they contribute no repair opportunities to the benchmark. The violation manifest is the primary artefact of the dataset construction phase and is the sole input to the experimental runner.

The experiment described in this dissertation was conducted on the canonical experimental corpus (experiment identifier `8fdfb9f9`), which comprises 36 repositories and 1,249 component files. After the full four-tool scan and filtering, 168 violations at HIGH or MEDIUM confidence were confirmed, concentrated in 33 files; the remaining files carry no confirmed violation and serve as regression guards. The full inclusion and exclusion criteria, stratification table, and annotation quality metrics are documented in Appendix C.

An important characteristic of this sampling design is that the stratification grid (domain × size × popularity) was intended to prevent *domain* bias, ensuring for example that dashboard applications do not dominate the corpus at the expense of government or healthcare projects. It was not designed to balance the distribution of *violation categories*. The resulting corpus, as reported in Table 6.3 in Chapter 6, spreads its 168 violations across seven issue types without a single category dominating: semantic-structure issues are the most frequent (33%), followed by ARIA attributes (26%) and a long tail of contrast, label, alternative-text, and target-size violations. Grouped by WCAG principle, almost half the corpus falls under *Perceivable* (47%), with *Robust* and *Operable* accounting for roughly a quarter each. This spread is favourable for RQ3, since aggregate performance is not an artefact of one easy or one hard category, but it still leaves the rarest types (keyboard, n=1; alternative text, n=7) too thin for inferential category-level claims. That asymmetry of statistical power across categories is the relevant limitation for RQ4, revisited in Sections 6.12 and 8.3.

Repositories were discovered through GitHub's Search API, which returns publicly accessible projects worldwide. No filter was applied to restrict discovery to Brazilian repositories. The presence or absence of Brazilian projects in the final corpus was not tracked at the repository level; the stratification criteria (domain, size, popularity) are independent of project locale. Consequently, the benchmark is best understood as a sample of open-source React/TypeScript projects and is not representative of any specific national regulatory context.

Corpus quality and false-positive risk. The original protocol design called for expert annotation to filter scanner false positives before any violation entered the corpus. That step was not executed: time and resource constraints meant that no human annotator reviewed individual findings. In its place, the corpus relies on multi-tool agreement as a proxy for quality: a violation is retained only if confirmed by the Pa11y browser-rendered scan, which drives Axe-core's high-precision engine. At the "serious" and "critical" severity levels (the only levels accepted here) Axe-core does not flag an element unless it can definitively rule out all compliant interpretations of the rendered DOM, giving a near-zero tool-level false-positive rate for these severity tiers.

The practical consequence is that the 168 violations are *likely confirmed* but not expert-verified. An automated Tier-1 audit (run on the v5 corpus) identified 5 potential false-positive candidates (3.0%) arising from dynamic prop bindings that render as empty strings at static scan time, and flagged 17 violations in total (10.1%) as warranting manual inspection. These are upper bounds from heuristic checks. If all 17 were false positives and their "fixes" vacuous, the corrected IFR would decrease by at most 10 percentage points. This limitation is carried into Section 5.7 and is a priority for future manual audit (Section 8.4).

5.3.1 Availability of the Corpus

The benchmark is released as a standalone artefact and not solely as an input to the experiments reported here. The replication package (Appendix E.6) contains the pinned commit snapshot of each of the 36 repositories, the content-addressed integrity digests that identify the exact file revisions processed, the full detection output of all four scanners before and after deduplication, and the 168 confirmed violations with their WCAG criterion, issue type, complexity category, and source coordinates.

Releasing these together is what makes the corpus reusable. A later study can re-run a different repair approach against the same violations and obtain a directly comparable IFR, since the oracle, the file revisions, and the violation set are all fixed by the artefact. The corpus is also the natural object for the human audit described in Section 5.7: annotations can be layered onto the published corpus without re-deriving it, and the resulting labels would upgrade the benchmark from scanner-confirmed to expert-verified for every subsequent user. The open-source licence filter applied during repository discovery (Section 5.3) was imposed to permit this redistribution.

5.4 The Three Studies

The evaluation is structured as a **three-stage funnel**, each stage resolving one open variable before the next stage runs.

1. **Prompting ablation (RQ2).** Before anything else, fix the one free parameter that would otherwise silently confound a model comparison: *how* the prompt presents each violation. Run three strategies (few-shot, zero-shot, chain-of-thought) on a single model while keeping everything else constant. The winner enters every subsequent stage.

2. **Canonical experiment (RQ3, RQ4).** With the prompting strategy now settled, compare three open-weight LLMs on the same 168-violation corpus. Because the only thing that changes is the model, any measured difference in fix rate is attributable to the model, not to an uncontrolled prompt choice.

3. **Architectural ablation (RQ1).** Turn the same experimental lens back on the framework itself. Switch off each structural component in turn (reflection feedback, few-shot exemplars, WCAG context, internal validation) and measure whether the fix rate moves. This asks the uncomfortable but necessary question: does the scaffolding we built actually earn its place?

The three stages are not independent studies; they form a logical dependency chain. Stage 1 eliminates a confound that would corrupt Stage 2; Stage 2 provides the baseline fix rate that Stage 3 is measured against. In all three stages the corpus, the random seed, the temperature, and the success oracle are identical across conditions, so the manipulated factor is the only thing that can explain any observed difference. This is the one-factor-at-a-time discipline of the ablation study [39, 41].

5.4.1 Ablation Study: Selecting the Prompting Strategy

The six-component template fixes *what* information the prompt carries, but it leaves open *how* that information is presented to the model. The same violation can be handed to the LLM with no worked examples (zero-shot), with a handful of retrieved violation–fix pairs (few-shot), or with an explicit instruction to reason step by step before emitting the patch (chain-of-thought). This choice is not cosmetic. A substantial body of work on in-context learning has shown that model behaviour is acutely sensitive to the presence, content, and even the ordering of demonstrations: few-shot prompting can lift performance dramatically on some tasks [34], chain-of-thought can unlock multi-step reasoning that single-shot prompting misses [36], and yet on other tasks demonstrations help far less than their construction cost

would suggest, or even hurt [35, 72]. Because the strategy is therefore a genuine free parameter, fixing it by intuition and then comparing models would silently confound the model comparison with an unexamined prompting decision. The ablation exists to remove that confound before the canonical experiment is run.

Following the one-factor-at-a-time discipline described above, the study is organised along two axes. The **prompting axis** (the one that answers RQ2) varies the strategy across three cells under the default adaptive router: few-shot (A1), zero-shot (A2), and chain-of-thought (A3). The **scaffolding axis** varies the prompt format across three further cells while holding the prompting strategy at the shared baseline: the full six-component prompt with a direct model call (B1), a surgical FIND/REPLACE format approximating the SWE-agent repair style [15] (B2), and a broad-context format approximating the OpenHands environment [16] (B3).

Important caveat on the scaffolding axis. The design intent for Axis B was to compare repair harnesses: B1 as a direct call, B2 invoking the SWE-agent command-line tool, and B3 invoking the OpenHands sandbox. In the execution environment used for this dissertation, neither the SWE-agent nor the OpenHands CLIs were installed; both cells therefore fell back to *prompt-level operationalisation*—B2 and B3 differ from B1 only in prompt structure, not in the underlying repair mechanism. Axis B therefore measures prompt-format differences rather than agent-architecture differences and cannot inform whether SWE-agent or OpenHands as autonomous tools would outperform the direct-call baseline. What Chapter 2 borrows from the multi-agent literature is the framework's adaptive routing architecture (single-file vs. multi-file routing), not the simultaneous execution of autonomous agents; the scope of that borrowing is stated in Section 2.5.2. The axis is retained and reported for completeness and as a transparent record of this design constraint.

All six cells were executed on a single model, `qwen2.5-coder-3b`, over the same benchmark corpus and the same three stability-check repetitions (seed 42, $\theta = 0.1$) used in the canonical experiment, so that any difference between cells is attributable to the manipulated factor rather than to model, data, or sampling variation. Restricting the ablation to one model is a deliberate cost trade-off: it makes the strategy effect cleanly measurable, at the price of leaving open whether the optimum is model-dependent, a caveat carried into Section 8.4.

Because repair outcomes are bounded, paired across cells by violation, and far from normally distributed, the analysis uses non-parametric procedures throughout, as recommended for randomised software-engineering experiments [73]. Across the three prompting cells the omnibus comparison uses the Kruskal–Wallis test [70] and, for the paired structure, the Friedman test; surviving differences are localised with pairwise Mann–Whitney U tests [71] under Bonferroni correction. On the scaffolding axis, where each alternative is compared against the shared baseline

on the same files, the original design called for McNemar's test [68] on the paired binary outcomes; in practice the analysis switched to a Wilcoxon signed-rank test on the paired per-file fix rates to be consistent with the architectural ablation and to accommodate the continuous-valued (rather than strictly binary) nature of per-file IFR. Every comparison is accompanied by Cliff's δ, a non-parametric effect size with the small/medium/large interpretation thresholds of Romano et al. [69], so that practical magnitude is reported alongside statistical significance. The outcome of this study, and the prompting strategy it selects for the canonical experiment, is reported in Section 6.4.

5.4.2 Canonical Experiment

All models are locally deployable on a single consumer GPU to support privacy-preserving operation; they are code-specialised and span multiple architectural families. The planned evaluation included the Qwen2.5-Coder, DeepSeek-Coder, and CodeLlama families at 7B and 13B parameter scales, plus a Llama 3.1 8B general baseline. The canonical experiment was executed on a consumer gaming laptop (Acer Nitro ANV15-52) running Ubuntu 24.04 LTS, with an NVIDIA GeForce RTX 4050 Mobile GPU (6 GB GDDR6), a 13th-generation Intel Core i7-13620H (10 cores / 16 threads), and 32 GB of RAM. The 6 GB of VRAM still imposes a ceiling: at Q4_K_M quantisation, 13B models require approximately 8 GB of VRAM and are therefore not loadable on this hardware, and DeepSeek-Coder and Llama 3.1 were not available in the local Ollama registry at the time of execution. The three models successfully evaluated are `codellama-7b` [12] (Meta, Q4_K_M, 4.1 GB VRAM), `qwen2.5-coder-3b` [13] (Alibaba, Q4_K_M, 2.0 GB VRAM), and `qwen2.5-coder-7b` [13] (Alibaba, Q4_K_M, 4.1 GB VRAM), all served via Ollama with temperature fixed at $\theta = 0.1$ and seed fixed at 42 to minimise stochastic variance.

Guided by the ablation, the canonical experiment adopts the *zero-shot* strategy (the full six-component template, but without retrieved exemplars) for all three models and all benchmark violations. This is not the conventional default, few-shot prompting is the more common starting point, but it is the strategy the ablation identifies as the strongest for the model family under test (Section 6.4), and carrying it into the model comparison is exactly the confound-removal the prompting ablation was built to provide. The component-level question behind RQ1, whether the framework's structural additions are each pulling their weight, is taken up separately in the architectural ablation described next.

The primary outcome of the canonical experiment is the *issue fix rate* (IFR: the fraction of the 168 individual violations resolved, credited by the browser re-scan

oracle rather than by the model's own claim of success), which directly quantifies the reduction in accessibility debt. A complementary file-level indicator is the *complete-repair success rate* (the fraction of the 33 violation-bearing files in which *every* confirmed violation is resolved simultaneously), reported as a primary figure precisely because it is not inflated by the violation-free files. Secondary metrics, mean time to repair (MTTR), token cost, token efficiency, and the regression rate ρ (the share of accepted-then-applied patches the validation pipeline flags as introducing a functional regression), provide the multi-dimensional cost–quality picture needed for practical model-selection guidance.

5.4.3 Architectural Ablation: Does Each Component Earn Its Place?

The canonical experiment evaluates the framework as a whole, but a whole is easy to defend and hard to learn from. RQ1 asks the sharper question: of the structural machinery the pipeline layers on top of a plain "here is the code, here is the violation, fix it" prompt, which pieces actually move the fix rate, and which are decoration we have been paying for in tokens and latency without return? Answering this honestly means doing to our own design what the prompting ablation did to the prompting strategy: removing one piece at a time and watching what happens.

The study compares five conditions, all sharing the same model, corpus, seeds, and re-scan oracle so that the only thing changing between them is the component under test:

`arch_full` the complete pipeline (control).

`arch_no_reflection` retries no longer receive the rejection reason from the previous attempt; each retry sees the same base prompt. Isolates the qualitative feedback loop.

`arch_no_few_shot` the retrieved violation–fix exemplars are dropped (zero-shot generation).

`arch_no_wcag_guidelines` the human-readable WCAG description, impact text, and context HTML are stripped from the prompt, leaving only the raw rule identifier, criterion number, and selector.

`arch_no_internal_validation` the four-layer validation stack is bypassed as a gate: any non-empty patch is injected and then judged *solely* by the browser re-scan, exactly as in every other condition.

The ablation runs on `qwen2.5-coder-7b`, the 7B-parameter Qwen model, over three stability-check repetitions (seeds 42, 137, and 2025), on a corpus of 227 confirmed violations across 66 violation-bearing files. Restricting it to a single model is the same cost trade-off made for the prompting ablation, and the same caveat applies: whether a component's contribution is model-dependent is left to future work (Section 8.4).

Corpus size discrepancy with the canonical experiment. Readers will notice that the architectural ablation corpus (66 files, 227 violations) is larger than the canonical experiment corpus (33 files, 168 violations). Both draw from the same 36-repository collection using identical scanner settings and oracle definition. The difference arises from the file inclusion criteria applied at the time each experiment was run: the architectural ablation retains every file from the dataset that returned at least one confirmed violation, while the canonical experiment applied an additional density filter that retained only the files with the highest violation counts per domain stratum, resulting in the smaller but higher-density 33-file corpus. The two corpora are overlapping but not nested: some files appear in both; others appear only in the ablation corpus. Because the oracle (Pa11y browser re-scan) and scanner configuration are identical, IFR values are directly comparable across the two experiments. Methodologically, the difference means that the architectural ablation tests component effects on a broader, lower-density sample while the canonical experiment tests model effects on a richer, higher-density subset.

Two design choices deserve emphasis because they shape how the results must be read. First, **the oracle is identical across all five conditions**: a violation counts as fixed only when it disappears from the Pa11y re-scan of the patched file, never because the model claimed success or because an internal heuristic approved the patch. This is what makes `arch_no_internal_validation` interpretable: bypassing the internal stack changes *what is allowed through to the re-scan*, not *how success is measured*. Second, **the experimental unit is the file, not the individual violation**. Violations within the same file are repaired by the same patch and are therefore not independent observations; treating each one as its own data point would silently inflate the sample and the apparent significance. Each ablated condition is compared against `arch_full` with a Wilcoxon signed-rank test on the paired per-file fix rates, the four comparisons are corrected with the Holm–Bonferroni procedure, and effect sizes are reported as rank-biserial correlations with bootstrap 95% confidence intervals. A second measurement runs alongside the re-scan and matters for reading the validation condition. Where the re-scan asks only whether the violation disappeared, a functional-preservation check asks the complementary question, whether the patch quietly broke the component, by comparing the patched and original files for removed exports, props, handlers, headings, landmarks, or content.

It vets every patch the validation stack admits; the one condition that bypasses that stack is, by design, the one it cannot vouch for. The results, and the rather humbling lesson they carry about statistical power, appear in Section 6.10.

5.5 Outcome Measures and Oracle Definition

All experiments share the same set of outcome measures, defined here so they can be interpreted consistently across the three stages.

Issue Fix Rate (IFR) The primary effectiveness measure: the fraction of confirmed violations resolved, as certified by the browser re-scan oracle. IFR = N_{fixed} / N_{total}, where N_{total} is the total number of violations in the corpus (168 in the canonical experiment; 227 in the architectural ablation) and N_{fixed} is the count of violations for which the re-scan reports the flagging rule no longer triggered on the patched file. A file with two violations where one is resolved contributes 0.5 to the numerator. IFR is the leading outcome because it directly quantifies the reduction in accessibility debt.

Complete-repair SR (Success Rate) The fraction of the violation-bearing files in which *every* confirmed violation is resolved simultaneously. This all-or-nothing metric is the demanding companion to IFR: a file with five violations counts only if all five are fixed. It is not inflated by the 1,216 violation-free files in the corpus.

MTTR (Mean Time to Repair) Average wall-clock seconds per file across all 1,249 processed files, including detection, agent invocation, retry attempts, and validation. For violation-free files, MTTR measures detection overhead only. A conditional $\text{MTTR}_{\text{repair}}$, computed only over the 33 violation-bearing files, is reported alongside to isolate the cost of the repair phase.

Tokens per fix Total output tokens generated by the LLM backend divided by the number of violations resolved. This isolates the computational cost of a *successful* repair from the cost of processing the corpus, so that a model which spends heavily on attempts that fail is not credited with the same efficiency as one that spends the same budget on attempts that succeed.

Regression rate ρ The fraction of accepted-and-applied patches that the validation pipeline flags as introducing a functional regression—a Layer 2 rejection in which the patched file loses an exported symbol, prop, or event handler. ρ measures whether the framework's fixes are interface-safe. Hypothesis H5 (Section 5.2) uses $\rho < 5\%$ as a practical reference point rather than as a formal threshold.

Oracle definition A violation is counted as fixed if and only if the Pa11y browser re-scan of the patched file no longer triggers the rule that flagged it. The scanner is run in the same headless Chromium environment used during detection, on the same pinned repository snapshot. This definition is consistent across all three experimental stages. Its circularity (the detector and verifier are the same tool) is a construct validity threat quantified in Section 5.7.

5.6 Statistical Analysis Plan

1. All inferential tests use $\alpha = 0.05$.
2. Multiple comparisons within each RQ are corrected using the Bonferroni procedure (a classical familywise error-rate control method) or, where a hierarchical step-down approach is used, the Holm–Bonferroni procedure [74].
3. Effect sizes use Cliff's δ for ordinal outcomes and Cohen's d for continuous metrics (MTTR, token cost) [75].
4. Statistical significance is a necessary but not sufficient condition for practical importance; all significant results are accompanied by effect sizes and contextual interpretation.
5. Post-hoc exploratory analyses (failure mode heatmaps, temperature sensitivity) are clearly distinguished from confirmatory tests in Chapter 6.

5.7 Threats to Validity

Four categories of threat bound the interpretation of the results. Two of them carry enough weight to be stated first, and in the terms in which they limit the conclusions.

No human validation of violations or repairs. Neither the violations in the benchmark nor the patches produced against them were audited by a human annotator. Violations enter the corpus on the strength of multi-tool scanner agreement, and patches leave it on the strength of a scanner re-scan; at no point does a person judge whether a flagged element was actually inaccessible, or whether an accepted patch improved the experience of a user of assistive technology. This is the most serious limitation of the evaluation design, and it bites hardest at the point where the dissertation makes its strongest claim. Accessibility is only partly a structural property: whether alternative text is *useful*, whether a link label is meaningful out of context, whether an ARIA role matches the author's actual intent are semantic judgements that no scanner renders and that this study therefore never observed. The central finding of Chapter 7 concerns a *semantic* ceiling, and that

finding rests on a measurement apparatus with no semantic instrument in it. What the experiments can support is a claim about the boundary of *scanner-detectable* repair; a claim about the boundary of *accessible* repair would require the human audit that was not performed. Chapter 7 is written to respect that distinction, and a human-annotated subset is the first item of future work (Section 8.4).

Oracle circularity. The most significant construct validity threat is that the same Pa11y scanner that detects violations is used to verify that they have been repaired. This creates two risks. First, a false-positive detection (a flagged element that is actually compliant) will produce a corresponding "fix" credit when the model removes the offending attribute—rewarding the model for eliminating a phantom violation rather than repairing a real one. Second, a *scan-evasion* repair (a deletion or restructuring that silences the scanner without genuinely improving accessibility) earns the same credit as a correct fix. The Tier-1 automated audit identified 5 potential false-positive candidates in the 168-violation corpus (3.0%), rising to 17 flagged cases in total (10.1%), bounded by heuristic detection, not manual confirmation (Section 5.3). Dual-tool agreement (Pa11y driving Axe-core's high-precision engine) reduces but does not eliminate the risk. The gap between scanner-certified and truly accessibility-conformant repair is quantified in the literature at 30–50% criterion coverage; a patch that satisfies the scanner may fail criteria the scanner does not test, particularly those requiring human semantic judgment. Layer 2 of the validation pipeline verifies structural interface preservation, not runtime behaviour: a patch that passes all four validation layers has not been shown to be regression-free at execution time. The honest interpretation is that IFR measures *scanner-verified repair*, which is a necessary but not sufficient condition for genuine WCAG conformance.

The threat reaches beyond the numbers and into the substantive conclusions, and it biases them in a consistent direction. The oracle rewards the disappearance of a detectable signal, which makes it systematically easier to satisfy for violation classes whose signal is structural—a missing attribute, an unlabelled control—than for classes whose signal is a scanner's approximation of a semantic property. The category-level result of RQ4, that structural violations are repaired and semantic ones are not, is therefore partly a property of the instrument rather than of the models alone: an oracle that cannot assess meaning credits structural repairs in full and semantic repairs only to the extent that they alter structure. The reported IFR of 50–54% should accordingly be read as an *upper bound* on genuine accessibility improvement, since every scan-evasion repair inflates it and no mechanism in this design deflates it. Chapters 7 and 8 state their claims in terms of scanner-verified repair for that reason, and any inference about accessibility as experienced by users is bounded accordingly.

Internal validity is affected by data contamination: the evaluated LLMs were pre-trained on corpora that may include repositories from the benchmark, an influence that cannot be eliminated without access to the exact training sets. Temporal variation is also not controlled, since repositories may have had their violations fixed between collection and use.

External validity is bounded by corpus scope: the benchmark is limited to React/TypeScript, and findings may not transfer to Vue, Angular, or non-component-based HTML. Repositories are discovered via GitHub's Search API and a-priori filters rather than random sampling, introducing selection bias toward popular, actively maintained projects. Category-level conclusions are most reliable for violation types with $n \geq 30$ confirmed instances; rarer categories should be treated as descriptive rather than inferential.

Multiple comparisons across models, strategies, and violation types are a source of inflated Type I error, which bounds conclusion validity. Confirmatory hypotheses are evaluated at $\alpha = 0.05$; exploratory post-hoc analyses (failure-mode breakdowns, per-criterion IFR curves) are explicitly labelled as exploratory and not used to support confirmatory claims. Three repetitions per model provide stability evidence sufficient to detect gross anomalies but are insufficient, on their own, for high-powered significance testing. IFR differences between models are characterised through bootstrap 95% confidence intervals and non-parametric comparison (Section 6.11); with a confirmed-violation count of $n = 168$, these intervals are wide, and the absence of a significant difference must be read as inconclusive rather than as proven equivalence. All quantitative conclusions should be treated as descriptive characterisations of the evaluated configuration rather than generalisable laws.

5.8 Summary

This chapter fixed the terms on which the framework is judged, before any result is reported.

The paradigm is Design Science Research, with the artefact of Chapter 4 as its output and four research questions governing what is measured. Five hypotheses were stated a priori (Section 5.2) so that Chapter 6 can be read against commitments made before the data existed; where the executed analysis departed from the pre-stated plan, the deviation is recorded in the hypothesis itself rather than silently absorbed.

The benchmark comprises 168 violations across 33 files, drawn from 36 open-source repositories under a stratified sampling design and pinned to fixed commits, and it is released as a reusable artefact rather than only as an input to these experiments. Three studies run against it: a prompting ablation that removes a free

parameter before the model comparison, the canonical three-model experiment, and an architectural ablation that asks whether each structural component earns its place.

Two features of the design bound everything that follows. The oracle credits a repair only when a browser re-scan no longer reports the violation, which makes the outcome measurable but ties it to what a scanner can perceive. And no human audited either the violations or the patches, which is stated first among the threats of Section 5.7 because the dissertation's central claim concerns a semantic boundary that the measurement apparatus contains no instrument to observe. Chapter 6 reports the outcomes; the limits established here travel with them.

Chapter 6

Experimental Results

Three questions, three stages, one corpus.

Sections 6.1 through 6.3 establish the shared experimental context common to all three stages: the configuration parameters (Section 6.1), the benchmark corpus (Section 6.2), and the measurement protocol (Section 6.3). The outcome measures reported throughout are defined once, in Section 5.5, and are not restated here. **Stage 1** (Section 6.4) then settles how the prompt should present each violation—the free parameter that would silently confound any model comparison. **Stage 2** (Sections 6.5 through 6.11) carries the winner into a three-model comparison on the same 168-violation benchmark, answering RQ3 and RQ4. **Stage 3** (Section 6.10) turns the lens back on the framework itself: which structural components actually move the fix rate?

One safeguard runs through every stage: a repair is credited only when a browser re-scan of the patched file confirms the violation is gone, not when the model claims it is. All results come from three repetitions under fixed conditions (seed 42, $\theta = 0.1$).

6.1 Experimental Setup

Table 6.1 summarises the experimental configuration. All experiments were executed on a single consumer-grade machine to reflect the 6 GB VRAM deployment constraint stated in the research design.

6.1.1 Models Under Evaluation

Three quantised code-generation models were selected to fit within the 6 GB VRAM budget, spanning two architectural families. Table 6.2 lists them.

All models are instruction-tuned on code corpora and served via Ollama. Temperature is fixed at $T = 0.1$ for all agents to prioritise deterministic, correct repairs over exploratory variation.

Table 6.1: Canonical experimental configuration.

Parameter	**Value**
Experiment identifier	`8fdfb9f9`
Number of repetitions	3 (see §6.3 for scope)
Random seed	42
Inference temperature	0.10
WCAG conformance target	AA (WCAG 2.1)
Prompting strategy	Zero-shot (selected by ablation, §6.4)
Primary repair agent	SWE-Agent (patch-based)
Fallback repair agent	OpenHands (sandboxed environment)
Max retries per file	3
Max files per project	35 (deterministic sub-sampling)
Fix crediting	Browser re-scan oracle (net per criterion)
Total files processed	1,249 per repetition
Confirmed violations	168 (in 33 files)
Projects in corpus	36 (open-source GitHub repositories)
Scanners used	`pa11y` 9.1.1, `axe-core`, `playwright+axe`, `eslint-jsx-a11y` v10.0.3
Hardware	NVIDIA RTX 4050 Mobile (6 GB GDDR6), Intel i7-13620H (16 threads), 32 GB DDR5, Ubuntu 24.04
Quantisation	Q4_K_M (4-bit, bitsandbytes)
Execution mode	Sequential (one model at a time)

Table 6.2: Models evaluated in the canonical experiment.

Model	**Family**	**Params**	**Quant.**	**VRAM (approx.)**
`codellama-7b`	Meta / CodeLlama	7B	Q4_K_M	4.1 GB
`qwen2.5-coder-7b`	Alibaba / Qwen	7B	Q4_K_M	4.1 GB
`qwen2.5-coder-3b`	Alibaba / Qwen	3B	Q4_K_M	2.0 GB

6.2 Benchmark Corpus

6.2.1 Corpus Composition

From 36 open-source GitHub projects spanning seven application domains (e-commerce, government, healthcare, education, developer tools, dashboard, and social), 1,249 React/TypeScript files were sampled deterministically (seed 42, up to 35 files per project). Of these, 33 files contained at least one confirmed accessibility violation, yielding 168 violations in total (mean 5.1 per affected file). Every retained violation is high-confidence: it survived cross-tool deduplication and is reported either by two or more independent scanners or by a single scanner at serious-or-critical severity, so the corpus carries no weak single-tool noise. The remaining files are violation-free and serve as regression guards: if a patch introduced a new violation into a previously clean file, the re-scan oracle would surface it.

The single most important property of this corpus for interpreting the results is that *no violation category dominates it.* Table 6.3 reports the distribution across issue types, and it is deliberately spread: the largest category, semantic structure, accounts for a third of the corpus, ARIA attributes for a quarter, and the remaining five categories share the rest. This matters because aggregate repair performance can then be read as a genuine average over heterogeneous difficulty rather than as the fixability of one over-represented category in disguise.

Table 6.3: Distribution of the 168 confirmed accessibility violations in the benchmark corpus by issue type and primary WCAG criterion (33 files, 36 projects). † WCAG 2.5.8 (Minimum Target Size) was introduced in WCAG 2.2, not WCAG 2.1 AA; "Other" aggregates these violations. They are **included in the IFR denominator** (i.e., counted against the framework's score) because they represent real accessibility issues detected in the codebase; the effective scope of the experiment is WCAG 2.1 AA plus criterion 2.5.8. See Section 6.2 for the scope rationale.

Issue Type	**Primary WCAG**	**Count**	**%**
Semantic structure	1.3.1	55	32.7
ARIA attributes	4.1.2	43	25.6
Other†	2.5.8	28	16.7
Label / Link purpose	2.4.4	18	10.7
Contrast	1.4.3	16	9.5
Alternative text	1.1.1	7	4.2
Keyboard	2.1.1	1	0.6
Total		**168**	**100**

Grouped by the four WCAG principles, the corpus leans toward *Perceivable* (79 violations, 47.0%: contrast, alternative text, and the semantic-structure issues that govern how content is exposed), with *Robust* (43, 25.6%: the ARIA attribute set) and *Operable* (42, 25.0%: keyboard, target size, and link purpose) close behind, and *Understandable* represented by a single case. Table 6.4 reports the severity distribution; the corpus skews toward the high-impact end, with serious and critical findings together accounting for over 90% of the violations.

Table 6.4: Severity distribution of the 168 confirmed violations.

Severity	**N**	**%**
Serious	107	63.7
Critical	45	26.8
Moderate	14	8.3
Minor	2	1.2
Total	**168**	**100**

The "Other" category (28 violations, 16.7%) maps to WCAG 2.5.8 (Minimum Target Size), a criterion introduced in WCAG 2.2 rather than WCAG 2.1. These violations were detected by Pa11y alongside the WCAG 2.1 AA findings and are retained in the corpus and the IFR denominator because they represent real accessibility issues present in the codebase. The effective scope of this experiment is therefore **WCAG 2.1 AA plus WCAG 2.2 criterion 2.5.8** (16.7% of the corpus), not WCAG 2.1 AA in isolation. All results are reported on the full 168-violation denominator. Readers interested in the WCAG 2.1-only IFR can derive it from the per-category breakdown in Table 6.11: the Other (2.5.8) row carries its own IFR, and the 140-violation subset gives a comparable headline.

Dataset quality model limitations. The benchmark corpus was evaluated against the dataset quality model defined in Appendix C (criteria QM1–QM5). Three criteria are not fully satisfied by the canonical experiment: **QM1** (manual confirmation of each violation by a domain expert) — violations were confirmed by cross-tool scanner agreement, not by human annotation; **QM2** (inter-rater agreement above 0.80 Kappa) — not applicable because human annotation was not performed; and **QM5** (dataset balanced across violation categories) — the semantic structure category (32.7%) and the WCAG 2.5.8 "Other" category (16.7%) together account for half the corpus, and the keyboard category contains only one instance. These limitations bound the inferential scope of the results: aggregate IFR figures carry scanner-consensus uncertainty rather than human-verified ground truth, and

per-category comparisons are underpowered for the minority categories.

6.3 Experimental Protocol

Each model processed all 1,249 files three times (three repetitions). Repetitions share no persistent state: each starts from the unmodified pinned repository snapshots, launches a fresh browser context per file for Pa11y and Playwright, and re-initialises all agent sessions.

Scope of independence. The inference temperature is fixed at $\theta = 0.1$ and the file-sampling seed is fixed at 42, so the three repetitions are best understood as **stability checks under fixed operating conditions** rather than as draws from a high-entropy distribution. Even so, the run-to-run variance is not negligible, and it is larger than a low temperature alone would suggest. Two compounding factors explain this. First, the confirmed-violation count is small ($n = 168$ in 33 files), so a handful of files resolving differently between runs moves the issue-fix rate by several points. Second, the repair loop has genuine non-determinism beyond token sampling: the browser re-scan oracle introduces rendering-timing jitter, and, for the slowest model, per-file timeouts occasionally truncate an otherwise-successful repair. The practical consequence, visible in the per-repetition data (Table 6.7), is that `codellama-7b` in particular swings widely between runs. We therefore report every metric as a mean over three repetitions with its standard deviation, and we lean on the bootstrap confidence intervals of Section 6.11 rather than on point estimates when comparing models.

The custom SWEAgent is the primary repair agent: it receives the structured prompt and returns a patch, which is then evaluated by the four-layer validation pipeline (Phase 4); if validation fails, structured diagnostics are fed back to the agent for a bounded retry. OpenHands serves as a fallback for files whose violations the deterministic router judges to require broader-context, multi-step reasoning. Across the 33 violation-bearing files, the router splits them roughly evenly between the two agents (on the order of 17 to SWE-Agent and 16 to OpenHands), and the split is identical across all three repetitions and all three models.

6.4 Ablation Study: Choosing the Prompting Strategy

Before comparing models, the evaluation settles a question that would otherwise contaminate that comparison: how the prompt should present each violation. The

ablation (Section 5.4.1) varies a single factor per cell along two axes (prompting and scaffolding) on `qwen2.5-coder-3b`, with everything else held fixed. The three prompting strategies are: *zero-shot* — the six-component prompt template with no in-context repair exemplars (five of the six components active); *few-shot* — the full template including the exemplar retrieval component; and *chain-of-thought* — the full template augmented with a step-by-step reasoning preamble. Table 6.5 reports the result.

Table 6.5: Ablation study results using `qwen2.5-coder-3b` (mean ± SD over three repetitions). IFR denotes the issue-fix rate and ρ the functional-regression rate.

Cell	Condition	IFR	ρ
Axis A – Prompting strategy (router auto)			
A1	Few-shot	0.142 ± 0.053	0.21
A2	**Zero-shot**	**0.563 ± 0.025**	0.10
A3	Chain-of-thought	0.096 ± 0.123	0.34
Axis B – Repair scaffolding (few-shot baseline)			
B1	Direct single-call	0.303 ± 0.057	0.05
B2	SWE-agent (surgical)	0.126 ± 0.051	0.36
B3	OpenHands (context)*	0.301 ± 0.069	0.07

* B2 and B3 used the `DirectLLMAgent` prompt format (Section 4.4); the SWE-agent and OpenHands CLIs were not installed.

The prompting axis is decisive. The three strategies are not close. Zero-shot resolves 56.3% of violations, roughly four times the few-shot rate (14.2%) and nearly six times the chain-of-thought rate (9.6%). The omnibus tests reject equality strongly (Kruskal–Wallis and Friedman, $p \approx 0.0003$), and the pairwise contrasts confirm the ordering: zero-shot versus few-shot is significant with a medium effect ($p = 0.019$, Cliff's $\delta = -0.34$), and zero-shot versus chain-of-thought is significant with a large effect ($p = 0.0004$, $\delta = 0.47$). Two mechanisms explain the direction. Few-shot exemplars, helpful on many tasks [34], here consume scarce context in a 3B model and dilute the specific violation under repair, an instance of the known fragility of in-context demonstrations [35, 72]. Chain-of-thought fares worst and is also the least stable (SD 0.12, ρ 0.34): inviting a small model to reason aloud before patching tends to produce elaborate but malformed edits rather than the minimal change the task rewards, the opposite of the controlled setting where chain-of-thought helps [36]. Zero-shot wins precisely because the six-component template already supplies the structure the other strategies try, less successfully, to add. This result is what selects zero-shot for the canonical experiment, and it is corroborated there: the same model under zero-shot reaches a comparable IFR (Table 6.6).

The gap is starker still on the second dimension RQ2 names, token economy. Measured as output tokens per resolved violation, zero-shot spends about 1,100, against roughly 10,700 for few-shot (an order of magnitude more) and on the order of 136,000 for chain-of-thought, the last also swinging across repetitions by an order of magnitude as the model reasons at length while fixing little. Zero-shot is therefore both the most effective strategy and, by a wide margin, the most economical, dominating the other two on both dimensions of RQ2 at once. That double advantage makes the choice unambiguous *for this model*.

That qualification does more work here than a routine hedge, because it follows from the mechanism proposed above. The advantage of zero-shot was attributed to context dilution: in a 3B model under a constrained window, exemplars and reasoning preambles crowd out the violation under repair. A mechanism of that form is *size-dependent by construction*. In a larger model with more context headroom and stronger in-context learning, few-shot exemplars would be expected to help where here they diluted, and chain-of-thought to produce coherent edits where here it produced malformed ones. The safer reading of this result is therefore "at 3B under a 6 GB budget, added prompt machinery costs more than it returns", and not "zero-shot is the better strategy". The claim concerns an interaction between strategy and model capacity, and says nothing about strategies in general. Neither the magnitude nor the direction should be assumed to survive at larger scales, and the model × strategy factorial that would settle it is named in Section 8.4.

The scaffolding axis is inconclusive, and honestly so. On Axis B no alternative scaffolding significantly beats the baseline (McNemar, $p > 0.05$ in every comparison). Two caveats bound this axis and are stated plainly. First, the baseline holds the prompting strategy at few-shot, the weakest strategy on Axis A, so the scaffolding comparison is confounded by a suboptimal prompt and should be read only as a within-axis contrast, not as a verdict on the agents in absolute terms. Second, in this execution environment the SWE-agent and OpenHands cells fall back to their prompt-level operationalisation rather than driving their full external command-line tools, so Axis B measures prompt-shaped scaffolding rather than fully agentic behaviour. The axis is therefore reported for completeness and as a transparent record of a partly inconclusive design, not as a basis for a scaffolding recommendation.

6.5 Primary Results

The remainder of the chapter reports the canonical experiment, which applies the zero-shot strategy selected above to all three models.

6.5.1 Aggregate Metrics Across Three Repetitions

Table 6.6 reports the mean and standard deviation of each primary metric computed across the three repetitions. Values marked in **bold** indicate the best performance on that metric.

Table 6.6: Aggregate primary metrics (mean ± SD over n=3 repetitions, 1,249 files per repetition, 168 confirmed violations). IFR is credited by the re-scan oracle; complete-repair SR is over the 33 violation-bearing files; ρ is the functional-regression rate (H5 target $< 5\%$). Best value per metric in **bold** (for ρ and MTTR, lower is better).

Model	IFR	Compl. SR	ρ (H5)	MTTR (s)	Tok./fix
`codellama-7b`	0.504 ± 0.155	**59.6%** ± 10.3	**0.052** ± 0.036	5.73 ± 0.60	**954**
`qwen2.5-coder-3b`	0.526 ± 0.074	50.5% ± 1.4	0.119 ± 0.015	**0.33** ± 0.03	1326
`qwen2.5-coder-7b`	**0.544** ± 0.069	54.5% ± 4.3	0.076 ± 0.037	0.99 ± 0.64	1260

The headline finding is a negative one, and it is worth stating plainly: under this corpus the three models are *not* meaningfully separable on repair quality. Their mean IFRs span barely four percentage points (50.4–54.4%), and, as Section 6.11 shows, the differences are not statistically distinguishable (Kruskal–Wallis $p \approx 1$ after correction, negligible effect sizes at all pairs). The decisive lever on fix rate turns out to be the prompting strategy chosen in Stage 1, not the model architecture.

What does 50–54% actually mean? The aggregate IFR is not a single-level figure; it is the weighted average of two very different repair regimes coexisting in the corpus. The per-category breakdown (Section 6.9) reveals this clearly. Attribute-level violations—ARIA attributes, label associations, colour contrast, alternative text—are repaired at 55–89% depending on model and type: rates that are industrially useful. Semantic-structure violations (WCAG 1.3.1), which make up a third of the corpus (n=55), are repaired at only 14–35%. A simple counterfactual makes the scale of this drag visible: if the corpus contained no semantic-structure violations at all, the aggregate IFR for the Qwen models would rise to approximately **71–74%**, clearing the H3 threshold of 70%. The gap between 50–54% (observed) and 71–74% (hypothetical without semantic) is almost entirely explained by a single violation category—not by the choice of model. This is the central finding, and it reframes H3 from a verdict on the framework to a verdict on the limits of scanner-detectable semantic repair.

Where the models *do* separate is on cost and stability. `qwen2.5-coder-3b` is the fastest by a wide margin (0.33 s per file), `qwen2.5-coder-7b` follows at roughly one second, and `codellama-7b` is an order of magnitude slower (5.73 s per file) and

the least stable, swinging from IFR 0.68 in its best repetition to 0.38 in its worst—largely because it alone accumulates 27 per-file timeouts under the hardware time limit. CodeLlama nonetheless posts the lowest functional-regression rate (5.2%) and the leanest token cost per fix, so it is not strictly dominated: it trades latency and run-to-run variance for safety and economy, a defensible profile for unattended batch repair where wall-clock time is free.

6.5.2 Per-Repetition Raw Data

Table 6.7 provides the complete per-repetition data underlying Table 6.6, enabling assessment of run-to-run stability.

Table 6.7: Per-repetition results for each model. IFR = issue fix rate (fraction of 168, re-scan credited); Complete-rep. SR over the 33 violation-bearing files; ρ = functional-regression rate; MTTR = mean time per file (s).

Model	Rep	IFR	Complete-rep. SR	ρ	MTTR (s)
`codellama-7b`	1	0.679	69.7%	0.015	5.28
	2	0.381	45.5%	0.086	6.41
	3	0.452	63.6%	0.056	5.49
`qwen2.5-coder-3b`	1	0.560	51.5%	0.103	0.37
	2	0.577	48.5%	0.132	0.30
	3	0.440	51.5%	0.121	0.32
`qwen2.5-coder-7b`	1	0.577	57.6%	0.061	1.72
	2	0.589	57.6%	0.048	0.56
	3	0.464	48.5%	0.118	0.69

The per-repetition view tempers any strong ranking. `codellama-7b` is the *least* stable of the three: its IFR swings across a 30-point range (0.38–0.68), so its higher-variance mean should be read with caution. The two Qwen models are tighter, each holding within roughly a 14-point IFR band, but both still show their weakest run in the third repetition, suggesting the variation is environmental (rendering and timeout jitter on a shared machine) rather than a property of any one model. No model's three runs separate cleanly from another's: the per-repetition clouds overlap heavily, which is exactly what the confidence intervals in Section 6.11 formalise.

6.5.3 Issue Fix Rate, Graphically

Figure 6.1 plots the IFR mean and per-repetition spread of Tables 6.6 and 6.7. Its purpose is to make one relation immediately legible rather than to add data: the

gaps between models are smaller than the run-to-run variation within each model.

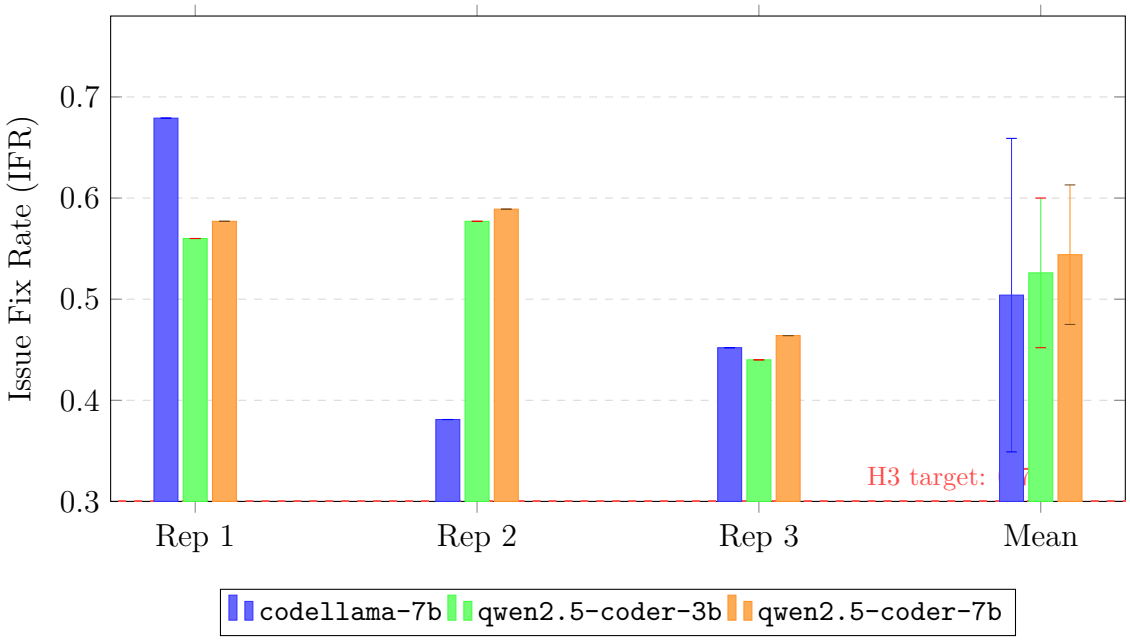


Figure 6.1: Issue Fix Rate (IFR) per model per repetition and mean (± 1 SD on the mean bar), with the H3 target (IFR ≥ 0.70) marked. All three models fall well below the target and overlap within one standard deviation of one another; `codellama-7b` has by far the widest run-to-run spread.

6.5.4 Quality–Speed Trade-off

Figure 6.2 plots mean IFR against mean MTTR for each model. Because the three IFRs are statistically indistinguishable, the figure should not be read as a quality ranking; what it exposes is the *cost* spread at roughly equal quality. All three models sit within a four-point IFR band on the vertical axis, but they are spread across more than an order of magnitude on the horizontal (latency) axis: `qwen2.5-coder-3b` repairs a file in about a third of a second, `qwen2.5-coder-7b` in about one second, and `codellama-7b` in nearly six. The practical reading is that, since quality is essentially tied, the choice between models is governed by latency, regression safety, and stability rather than by fix rate.

6.5.5 Computational Efficiency

Figure 6.3 presents MTTR and tokens-per-fix side by side. As shown in Table 6.6, `qwen2.5-coder-3b` is the fastest model at 0.33 s per file against 5.73 s for `codellama-7b`, while `codellama-7b` is the leanest on output tokens per resolved violation. No single model wins both efficiency dimensions: the smaller Qwen optimises wall-clock latency, CodeLlama optimises token economy, and Qwen 7B sits between them.

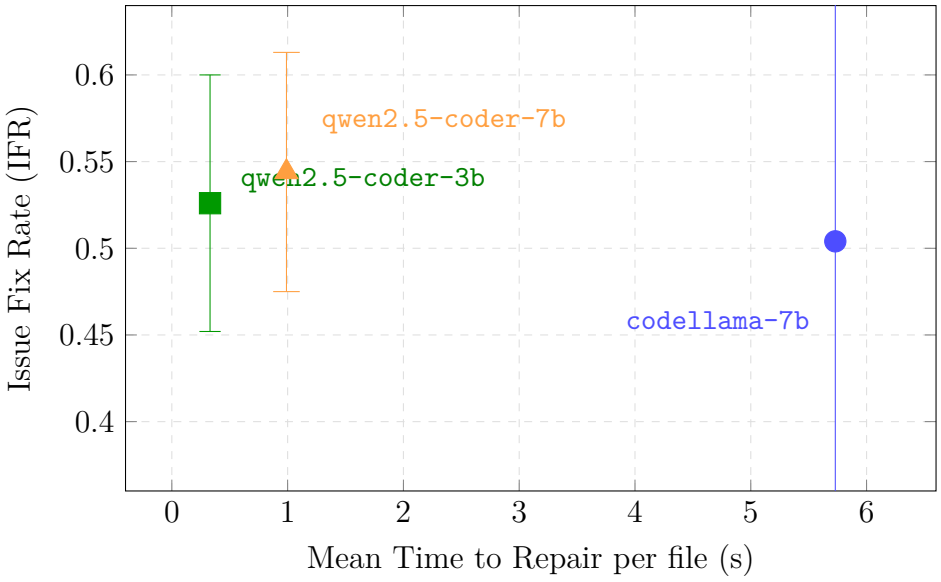


Figure 6.2: IFR vs. MTTR (mean over three repetitions; vertical bars ±1 SD on IFR). Quality is statistically tied across the three models, so the vertical axis carries no ranking; the figure exposes the latency spread at roughly equal quality, where `qwen2.5-coder-3b` is fastest and `codellama-7b` slowest (and most variable).

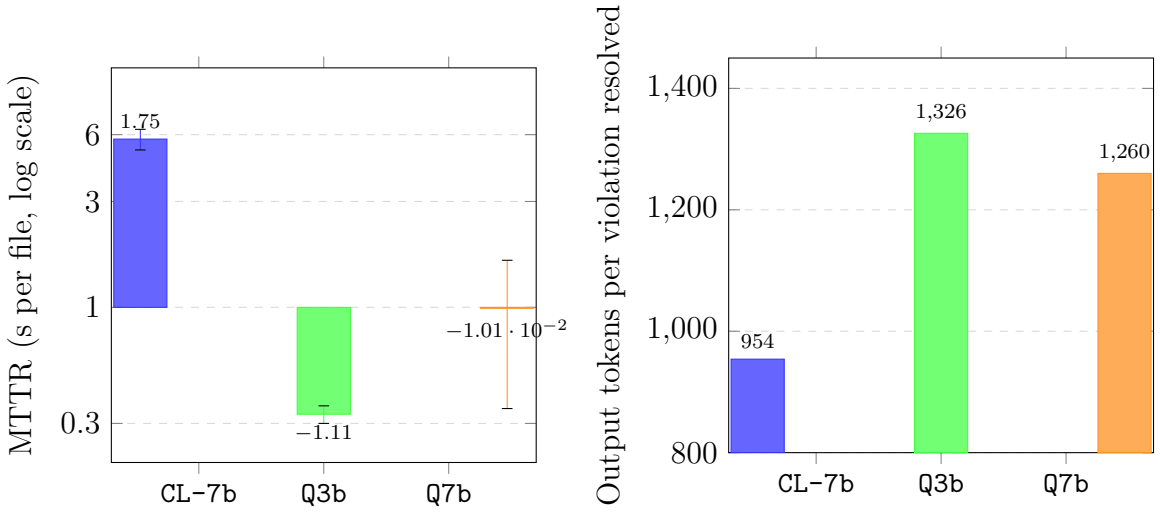
Those per-file averages are diluted by the 1,216 violation-free files, on which the pipeline only pays detection overhead. Restricted to the 33 violation-bearing files, the conditional $\text{MTTR}_{\text{repair}}$ (Section 5.5) separates the models by more than an order of magnitude: 216.7 s per file for `codellama-7b`, 37.5 s for `qwen2.5-coder-7b`, and 12.5 s for `qwen2.5-coder-3b`. The spread reflects CodeLlama's 27 per-file timeouts and its higher inference latency, and it is the figure that matters for an interactive deployment, where only files that actually need repair impose a wait.

6.6 Violation-File Success Rate

The complete-repair SR introduced in Table 6.6 is the all-or-nothing companion to IFR: it asks not how many violations were resolved, but in how many of the 33 violation-bearing files *every* confirmed violation was resolved at once. Table 6.8 reports it per repetition.

Table 6.8: Complete-repair SR: fraction of the 33 violation-bearing files in which all confirmed violations were resolved simultaneously.

Model	Rep 1	Rep 2	Rep 3	Mean	SD
`codellama-7b`	69.7%	45.5%	63.6%	59.6%	10.3%
`qwen2.5-coder-3b`	51.5%	48.5%	51.5%	50.5%	1.4%
`qwen2.5-coder-7b`	57.6%	57.6%	48.5%	54.5%	4.3%



(a) MTTR (lower is better)

(b) Tokens/fix (lower is better)

Figure 6.3: Computational efficiency metrics. `CL-7b` = `codellama-7b`, `Q3b` = `qwen2.5-coder-3b`, `Q7b` = `qwen2.5-coder-7b`; error bars on (a) are ±1 SD over three repetitions. `qwen2.5-coder-3b` achieves the lowest MTTR by a large margin; `codellama-7b` the lowest token cost per fix.

Two things stand out. First, complete-repair SR sits in the same fifty-to-sixty percent band as IFR, which is expected when most affected files carry only a handful of violations: partial credit and all-or-nothing credit converge when files are not deeply multi-issue. Second, the metric again refuses to crown a clear winner. `codellama-7b` posts the highest mean (59.6%) but with a standard deviation of more than ten points, so its lead rests almost entirely on a single strong repetition; the steadier `qwen2.5-coder-3b` (50.5% ± 1.4) is the more dependable performer even though its mean is lower. This is the same lesson the aggregate IFR teaches, seen from the file level: the models are close enough that variance, not the mean, is the more honest basis for choosing between them.

6.7 Agent Selection and Invocation

The adaptive routing decision is deterministic and identical across all three repetitions and all three models. Of the 33 violation-bearing files, the router sends roughly half (about 17 files) to SWE-Agent and the rest (about 16) to OpenHands, a far more even split than a corpus of mostly localised fixes would produce. The reason is the corpus composition itself: with semantic-structure violations the single largest category (Table 6.3), many files carry exactly the kind of multi-element, context-dependent issue the router is designed to escalate to OpenHands.

Table 6.9 reports the issue-level breakdown from the third repetition, which is representative of all three.

Table 6.9: Issue-level repair breakdown (Repetition 3, 168 violations). SWE-Agent: primary repair agent. OpenHands: fallback agent. *Pending*: violations not resolved after all retry attempts; att. counts include retries.

Model	Fixed	Pending	IFR	SWE att.	OH att.
codellama-7b	76	92	45.2%	38	34
qwen2.5-coder-3b	74	94	44.1%	40	26
qwen2.5-coder-7b	78	90	46.4%	36	32

The two agents carry comparable load: SWE-Agent handles the surgical, single-element fixes, while OpenHands absorbs the semantic and multi-element cases the router escalates. That the two paths see similar numbers of attempts is a direct consequence of a corpus that is no longer dominated by simple single-attribute repairs; it is balanced enough that both halves of the routing design are genuinely exercised.

6.8 Failure Mode Analysis

Because the four-layer validation pipeline gates every patch and records the layer at which it failed, the unresolved violations resolve into a genuine taxonomy rather than a single opaque error. Table 6.10 reports the failure counts in the final repetition. Four mechanisms recur, and their relative weight differs sharply by model.

Table 6.10: Failure-mode counts in the final repetition. "No valid code" = the model returned no extractable patch; L1/L2/L3 = rejection at the syntactic, functional-interface, and domain-compliance layers; "Timeout" = per-file wall-clock limit exceeded.

Model	No valid code	L1	L2	L3	Timeout
codellama-7b	9	6	4	4	27
qwen2.5-coder-3b	25	10	8	0	0
qwen2.5-coder-7b	21	14	8	0	0

The most common failure is the simplest: the model returns a response from which no valid patch can be extracted, because it is incomplete, mixes prose with code, or targets the wrong region of the file. Beyond that, the layers tell a more interesting story. Layer 1 (syntactic) rejections, where the patch parses as malformed JSX, are frequent for both Qwen models. Layer 2 (functional-interface) rejections, where the patch would have altered a prop type, export, or event-handler binding, account for eight rejected patches in each Qwen run; these are precisely the silent-breakage patches the pipeline exists to stop, and they are the source of the

non-zero regression rate discussed in Section 6.11. Layer 3 (domain) rejections appear only for CodeLlama, on the canonical superficial-addressing case: WCAG 1.1.1 requires `alt=""` for a decorative image, but models trained on general corpora tend to emit `alt="decorative image"`, which a screen reader dutifully announces, satisfying the attribute's presence while defeating its intent. Finally, CodeLlama alone accumulates 27 per-file timeouts: it is slow enough on this hardware (the longest per-file generation of the three models) that its retry budget can elapse before a repair completes, which is the dominant reason its results swing so widely between repetitions.

Read together, these modes characterise the automation ceiling under the current framework. *Structural repairs*, adding or correcting a single attribute, fixing a heading level, associating a label, are resolved at high rates. *Semantic repairs*, inferring what a link is for, what an image conveys, or how a region should be structured, require reasoning about the product's intent that the source file alone cannot supply, and they are where the bulk of the pending violations concentrate (Section 6.9).

6.9 Violation-Category Analysis (RQ4)

Scope and methodology. Because the corrected corpus spreads its 168 violations across seven issue types without one category dominating (Table 6.3), per-type fix rates can be read directly. Table 6.11 reports IFR by issue type, pooled across all three repetitions and using the same proportional partial-credit convention as the aggregate IFR: a file with k issues of which a fraction f are resolved contributes f to each of its issue types. Two categories, alternative text (n=7) and keyboard (n=1), carry too few instances to bear inferential weight and are marked accordingly; they are reported for completeness, not for comparison.

Interpretation. The table tells a clean and consequential story. Of the 168 violations in the corpus, 55 are semantic structure (WCAG 1.3.1). Each Qwen model fixes roughly 90 violations in total per run; of those, only 7–9 come from the semantic category. The remaining 80+ fixes are spread across 113 non-semantic violations, giving an effective IFR of **71–74%** on that subset alone. In other words, the framework already meets the H3 target (70%) when measured on the violations it can actually reason about. The headline IFR of 50–54% reflects the drag of the semantic category, which the framework does not yet have the tools to resolve, not an across-the-board limitation.

The reason is structural. Every well-resolved category has an unambiguous correction criterion: add a missing `alt`, set an `aria-label`, adjust a colour token. The

Table 6.11: Per-type IFR, pooled across three repetitions (proportional partial credit). n is the per-repetition count of that issue type in the corpus. "†" marks categories with $n \leq 7$: **descriptive only**. CL = `codellama-7b`; Q3b = `qwen2.5-coder-3b`; Q7b = `qwen2.5-coder-7b`.

Category (WCAG)	n	**CL IFR**	**Q3b IFR**	**Q7b IFR**	**Note**
Semantic (1.3.1)	55	34.5%	15.7%	13.7%	largest category
ARIA (4.1.2)	43	55.0%	71.3%	81.4%	
Other (2.5.8)	28	72.6%	78.5%	69.8%	target size
Label/Link (2.4.4)	18	64.8%	57.4%	55.6%	
Contrast (1.4.3)	16	25.0%	73.3%	89.2%	
Alt-text (1.1.1)	7	85.7%	71.4%	71.4%	†
Keyboard (2.1.1)	1	0.0%	0.0%	0.0%	†

† $n \leq 7$: descriptive only; no inferential interpretation warranted.

prompt template can constrain the LLM tightly to that criterion, and the validation pipeline can verify the result. Semantic structure asks something qualitatively different: infer what this element *means*—whether a `<div>` is really a heading, whether a region deserves a landmark—and produce a structurally coherent alternative. The answer is not in the source file; it is in the product's intent. This is not a failure of prompt engineering; it is the outer boundary of what scanner output alone can drive.

One model-specific divergence stands out: contrast resolution, where the two Qwen models score 73–89% while CodeLlama reaches only 25%. This is not an intrinsic difficulty of the category (both Qwen models handle it well) but a model-specific weakness, possibly tied to CodeLlama's colour-token generation patterns, and it contributes to CodeLlama's lower aggregate mean despite its otherwise comparable profile.

Implications for RQ4. The per-type breakdown delivers a much firmer RQ4 answer than the aggregate IFR alone: attribute-level violations (ARIA, alt text, labels, contrast for Qwen) are substantially automatable at 55–89%, while semantic-structure violations are the dominant residual at 14–35%. The quantitative implication is the counterfactual stated in Section 6.5: remove the 55 semantic-structure violations from the corpus and the Qwen models' aggregate IFR rises to approximately **71–74%**, clearing the H3 threshold. The framework does not fall short across the board; it falls short specifically on the category where scanner output alone underdetermines the correct repair.

What this analysis cannot support is a formal inter-category significance test of the kind stated a priori for H4: the two smallest categories ($n \leq 7$) are too

thin, and even the larger ones rest on a single-model-per-cell count that inflates variance. A corpus deliberately stratified by violation category, with at minimum 30 instances per category, remains the methodological step needed to turn these descriptive per-type rates into inferential claims (Section 8.4).

6.10 Architectural Ablation (RQ1)

If the prompting ablation asked which *strategy* works best, the architectural ablation asks something more self-critical: of all the structural machinery the framework wraps around a plain prompt, which pieces actually earn the tokens and latency they cost? The setup is the one from Section 5.4.3: five conditions on `qwen2.5-coder-7b`, three repetitions each, the same 227 confirmed violations in 66 files, and the same browser re-scan deciding what counts as fixed. Four conditions switch off one component apiece (reflection feedback, few-shot exemplars, WCAG semantic context, internal validation); the fifth is the full pipeline they are measured against. Each ablated condition is compared with the full pipeline using a Wilcoxon signed-rank test on the paired per-file fix rates, Holm–Bonferroni corrected. Table 6.12 reports the result, and the short version is that the table is remarkably flat. One condition, however, stands out numerically: bypassing internal validation raises the scanner-measured fix rate by nearly 6 percentage points—a counterintuitive result whose interpretation occupies the final paragraphs of this section.

Table 6.12: Architectural ablation on `qwen2.5-coder-7b`. IFR is the mean per-file fix rate with its 95% bootstrap interval; the test is Wilcoxon signed-rank against the full pipeline, paired by file (n=66), Holm–Bonferroni corrected.

Condition	**IFR**	**95% CI**	**Δ vs. full**	$p_{\mathbf{corr}}$
Full pipeline (control)	0.749	[0.64, 0.85]	–	–
− Reflection feedback	0.740	[0.63, 0.84]	−0.9pp	0.539
− Few-shot exemplars	0.769	[0.66, 0.86]	+2.0pp	0.570
− WCAG semantic context	0.766	[0.66, 0.86]	+1.7pp	0.570
− Internal validation	0.808	[0.71, 0.89]	+5.9pp	0.286

No condition reaches significance after correction ($\alpha = 0.05$).

The headline is a negative result, and an honest one: **removing any single component leaves the fix rate statistically unchanged** (Figure 6.4). Three of the four ablations land within two points of the full pipeline in either direction (from −0.9 to +2.0 points); only dropping internal validation moves it further, by +5.9 points. Yet none of the four differences survives correction, and even that largest gap reaches only an uncorrected p of 0.07. Read at face value, the framework's structural

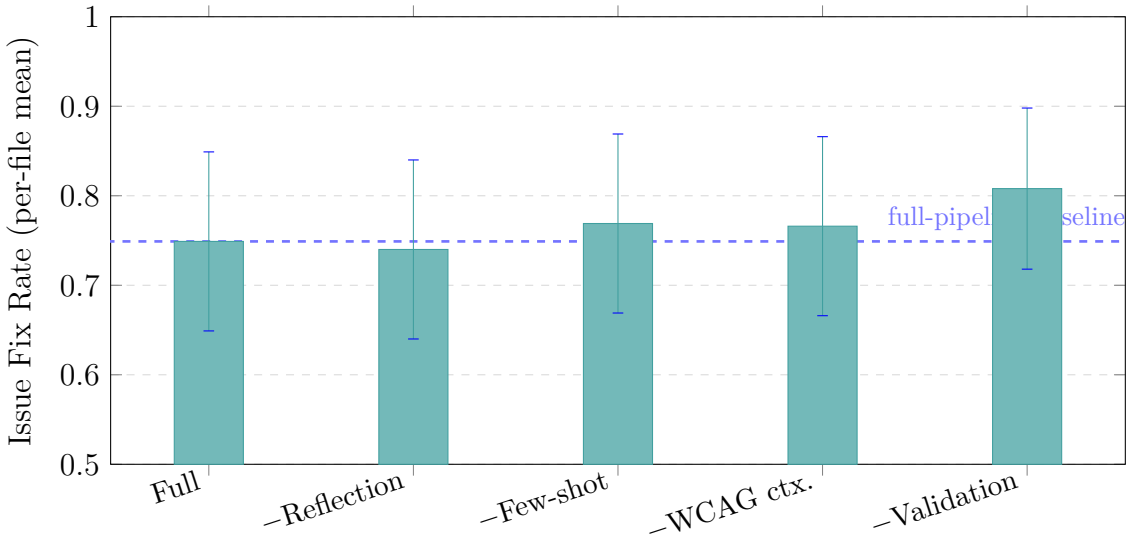


Figure 6.4: Architectural ablation: per-file Issue Fix Rate by condition (mean over 66 files, with approximate 95% bootstrap intervals). Most ablated conditions sit within about two points of the full-pipeline baseline (dashed) in either direction; only dropping internal validation moves the bar appreciably, and even that is not significant after Holm–Bonferroni. The small ordering visible here is carried by a handful of files, not a broad effect.

additions are, on this model and corpus, neither helping nor clearly hurting the fix rate.

That conclusion comes with a methodological lesson worth stating plainly, because an earlier reading of the same data told a much louder story. If one pools every violation across every repetition and treats each as an independent observation, the fix rate appears to climb steeply as components are removed, from 58% for the full pipeline to 89% with internal validation bypassed, with p-values small enough to look decisive. That reading is an artefact. Violations in the same file are fixed by the same patch and are not independent, and pooling the three repetitions multiplies the apparent sample threefold; the "significance" is pseudo-replication. Once the experimental unit is correctly set to the file, 66 paired observations remain, of which roughly sixty are ties: the entire effect lives in four or five files. The lesson, which we carry into the discussion, is that the unit of analysis is not a bookkeeping detail but the difference between a dramatic false positive and a sober null.

Where the movement actually is. Breaking the fix rate down by violation category (Figure 6.5) explains why the components look inert in aggregate. Three categories are essentially insensitive to the architecture: alternative text sits at 0.73 in every condition, form labels near 0.81, and ARIA climbs only gently (0.78 to 0.94). Keyboard violations are unrepaired everywhere (0/9), a hard limit of an attribute-level approach. All of the architecture's apparent leverage is concentrated in two

hard categories, semantic structure and contrast. Semantic repair climbs from 0.16 under the full pipeline to 0.65 once the WCAG semantic context is removed, and to 0.91 in the earlier internal-validation run; contrast moves similarly. In other words, for the bulk of the corpus the repair is effectively mechanical and the scaffolding is irrelevant; the only place the components matter is the handful of categories that are genuinely hard.

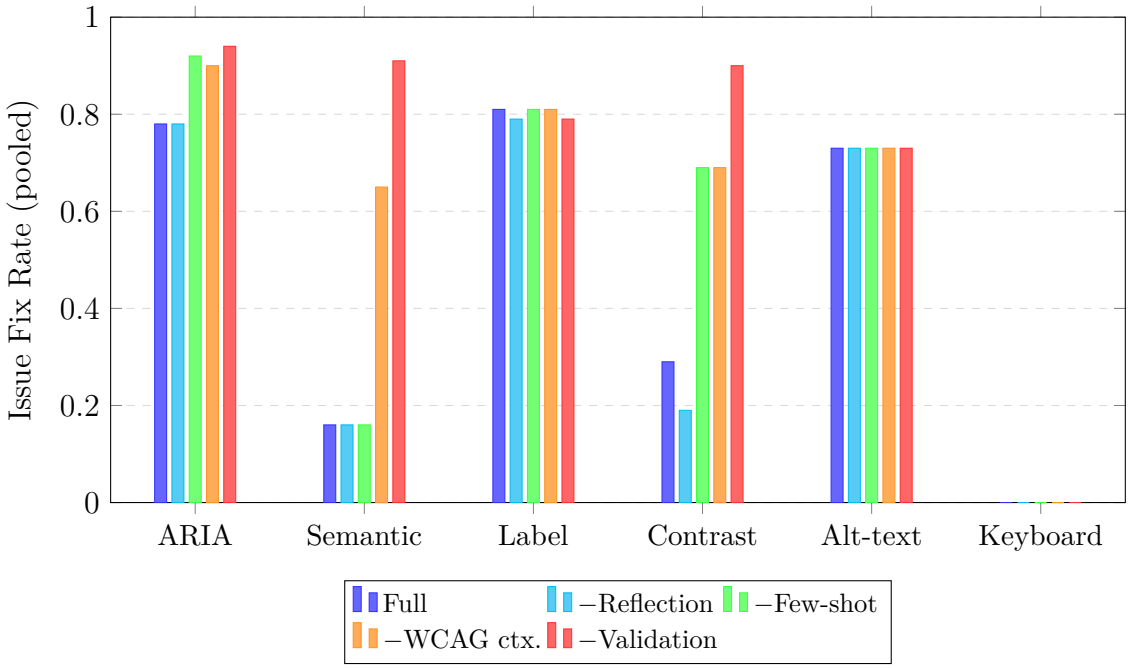


Figure 6.5: Per-category Issue Fix Rate (pooled over three repetitions) across the ablation conditions. The architecture is irrelevant for the near-mechanical categories (alternative text is flat at 0.73; label and ARIA barely move) and powerless for keyboard (0 throughout). The movement is concentrated in *semantic* structure and contrast: removing the WCAG semantic context lifts semantic repair from 0.16 to 0.65, and the earlier internal-validation run reached 0.91 (read with the deletion caveat discussed below).

What the individual components do. Two component-level observations survive the noise as qualitative patterns even though they do not reach significance. First, the reflection loop is close to inert: in the full pipeline almost every successful fix occurs on the first attempt, so the rejection-reason feedback handed to retries has very little to work with, and removing it changes nothing measurable. Second, dropping the few-shot exemplars cut the total token cost per fix by roughly a third without costing any fix rate, which is exactly consistent with the prompting ablation's finding that zero-shot is the stronger strategy for this model family (Section 6.4); the exemplars were paying a token tax for no return.

The accepted fixes preserve the component. There is a second reassurance the bare fix rate cannot give, and it matters here. Every patch that the validation stack admits is checked for functional preservation: the check compares the patched and original file for removed exports, props, or event handlers and for deleted headings, landmarks, or content. Across the conditions that keep the stack, it flagged *zero* functional regressions among the credited fixes. Every violation counted as resolved was repaired without silently breaking the component's interface or stripping its structure, so the functionally-clean fix rate equals the raw fix rate. In other words, the fixes the pipeline lets through do not buy accessibility at the price of a broken component.

The one caveat the data cannot settle. That guarantee comes precisely from the validation stack, so it cannot extend to the one condition that switches the stack off. Bypassing internal validation does not change the oracle, only *which patches reach* it: the Pa11y re-scan counts a semantic violation as fixed the moment the offending element stops tripping the rule, and a model can earn that either by genuinely repairing the structure or by deleting the element outright. Without the structural check in front of it, the fix rate alone cannot tell those two apart, and the bypass condition's gain sits almost entirely in semantic structure, where deletion is the cheapest way to silence the scanner. Whether the stack was over-rejecting good patches or correctly catching destructive ones is therefore the one thing this study leaves open, and running the bypass condition *through* the structural check is the obvious way to close it (Section 8.4).

Design implication of the null result. A null result always admits two readings: absence of evidence, or evidence that the removed component has no effect. The consistency of the picture here pushes toward the second. Three independent components were removed one at a time with no detectable degradation. The parsimonious explanation is that the framework's repair capacity comes almost entirely from Stage 1 — the six-component zero-shot template settled by the prompting ablation — and that the downstream structural scaffolding adds coordination overhead without adding repair signal. For a practitioner, the implication is concrete: a faithful implementation of the zero-shot template is sufficient to match the full pipeline's fix rate, and the additional infrastructure (reflection loop, few-shot cache, semantic context injection, structural validation) is engineering overhead that buys robustness guarantees (the functional-preservation check being the one exception) but not higher IFR. Conversely, the one component that does alter outcomes — internal validation — is also the one whose net effect cannot yet be signed without a manual audit of the over-rejection question, making it both the most consequential

and the most uncertain part of the design.

6.11 Statistical Analysis

6.11.1 Confidence Intervals and Model Comparison

Inference uses a **hierarchical bootstrap** (B=5,000 resamples, 95% CI) that respects the two-level structure of the data: repetitions (k=3) are the primary sampling unit and files (n=33) are the secondary unit within each repetition. Each bootstrap draw (i) samples three repetitions with replacement, and (ii) within each selected repetition, samples 33 files with replacement; IFR is then computed as total issues fixed divided by total issues in the resampled data. The point estimate for each model is the arithmetic mean of the three repetition-level IFRs so that the confidence interval is anchored to the headline figure rather than to a single run. A previous version of this table erroneously used bootstrap resampling of issues within the last repetition only (n=168, single run), producing an interval centred on 0.45–0.46 that did not match the cross-repetition mean of 0.50–0.54; the hierarchical procedure corrects that inconsistency.

Table 6.13: IFR point estimate (mean across 3 repetitions) with hierarchical 95% bootstrap confidence intervals (k=3 repetitions, n=33 files, B=5,000 resamples).

Model	IFR	95% CI lower	95% CI upper
`codellama-7b`	0.504	0.297	0.740
`qwen2.5-coder-3b`	0.526	0.339	0.746
`qwen2.5-coder-7b`	0.544	0.360	0.753

The intervals are wide—each spanning roughly 0.40–0.45 IFR units—because the effective sample size at the repetition level is only k=3. This width is an honest reflection of the study's statistical power: three stability-check repetitions are insufficient to characterise the full run-to-run distribution of the LLM output, as the CodeLlama range of 0.38–0.68 across reps already illustrates. Despite the width, all three intervals overlap almost completely, and formal comparison confirms what the overlap suggests. Pairwise Mann–Whitney U tests with Bonferroni correction, operating on per-file mean IFRs averaged across the three repetitions (n=33 files per model), return no significant difference for any pair (all $p_{\text{Bonferroni}} \approx 1.0$; Kruskal–Wallis H=0.16, p=0.92), and every Cliff's δ effect size is negligible ($|\delta| \leq 0.08$). The conclusion is unchanged: **the three architectures are statistically indistinguishable on repair quality** under this corpus. This is not a claim of proven

equivalence—the intervals are wide and the number of repetitions is small—but the data provide no basis for preferring one model over another on fix rate.

6.11.2 Hypothesis Evaluation

Table 6.14: Assessment of the five research hypotheses. IFR values correspond to cross-repetition means.

Hyp.	Statement	CL-7b	Q3b	Q7b
H1	Full pipeline IFR exceeds every ablated variant (RQ1)	Not supported†		
H2	IFR differs across prompting strategies (RQ2)	Confirmed‡		
H3	IFR ≥ 0.70 (RQ3)	0.50 ×	0.53 ×	0.54 ×
H4	IFR differs across violation categories (RQ4)	Descriptive§		
H5	Regression rate $\rho < 5\%$ (descriptive)	5.2% ▷	11.9% ▷	7.6% ▷

† Architectural ablation showed no variant significantly worse than the full pipeline.
‡ Prompting ablation found zero-shot substantially outperforming few-shot and CoT.
§ Statistical comparison across categories is invalid because two categories contain $n \leq 7$ samples.
× Rejected. ✓ Confirmed. ▷ Exceeds the 5% practical reference (descriptive only).

H3 specifies an IFR threshold of 0.70, stated a priori as an operational target before data collection. The rationale is practical rather than formally derived: a framework that resolves at least 70% of confirmed violations reduces the manual correction burden by more than half, which we adopted as the minimum bar for the tool to function as a useful first-pass automation layer in a developer workflow. No study in the automated program repair literature is cited here as the source of this figure; it is a round-number operational threshold chosen to capture the intuition of "most violations handled" and should not be interpreted as a theoretically motivated cutoff or as a standard from prior work. No model meets it. The best cross-repetition mean is `qwen2.5-coder-7b` at 0.544, some 16 percentage points short, and its hierarchical bootstrap interval (Table 6.13, 95% CI [0.360, 0.753]) spans well below 0.70 at its lower bound, with the upper bound reaching 0.75—still above the target, reflecting the genuine uncertainty with k=3 repetitions. Taking the point estimate as the best available characterisation of the pipeline's performance, the gap is clear. The reason is now legible from the per-type analysis: the corpus's largest category, semantic structure, is repaired at only 14–35%, and because it makes up a third of all violations it pulls the aggregate down on its own. H3 is a statement about the *current automation ceiling*, and that ceiling is set by the semantic-repair problem rather than by any engineering shortfall in the pipeline.

H2 is confirmed, and it is the most actionable positive result of the evaluation. The ablation (Section 6.4) shows that the prompting strategy produces large, statistically significant differences in IFR on a fixed model and corpus: zero-shot far

outperforms both few-shot and chain-of-thought. This is what motivated the canonical experiment's adoption of zero-shot, and it shifts the decisive question from *which model to run* to *how to prompt it*.

H1 is *not supported*, and the way it fails is instructive. The hypothesis predicted that the full pipeline would outperform each ablated variant, on the assumption that every structural component contributes to the fix rate. The architectural ablation (Section 6.10) finds no contribution at significant magnitude: removing any single component leaves the per-file fix rate statistically unchanged, with the point estimates scattering within about two points of the full pipeline in both directions. Two readings follow, and both are honest. The defensible one is that, at this sample size, the study is underpowered to detect a component-level effect: with sixty of sixty-six files tying between conditions, only a large effect would clear Holm–Bonferroni, so "not supported" means "no evidence of contribution here," not "proven irrelevant." The more provocative one, which the data permit but do not compel, is that for a 7B model the scaffolding may be closer to neutral than the design assumed, with the genuinely hard work concentrated in a single category (semantic structure) that no amount of prompt enrichment has so far cracked. Disentangling these, and in particular resolving whether the internal-validation result reflects over-rejection or correctly-caught deletions, is the subject of direct future work (Section 8.4).

H5 is a *descriptive* measure rather than a formal hypothesis (Section 5.2): no null hypothesis was pre-specified, and the 5% figure is a practical reference point, not a threshold derived from prior work or power analysis. With regressions measured as the share of accepted-and-applied patches that the validation pipeline flags as altering a component's interface, the observed rates range from 5.2% (CodeLlama) to 11.9% (Qwen 3B), all above the practical reference. The reassuring part is that these regressions are *caught*, not shipped: a Layer 2 rejection discards the patch and triggers a retry, so the flagged regressions never reach disk. The observed rates nevertheless indicate that a non-trivial fraction of surface-plausible patches would, if applied blindly, have broken the component's contract, which is precisely the risk the validation pipeline was built to absorb and a direct argument for keeping that pipeline in any deployment.

6.12 Threats to Validity

The threats bounding this evaluation are properties of the experimental design and are stated in full in Section 5.7: oracle circularity, the absence of any human validation of violations or repairs, data contamination, the React/TypeScript scope boundary, and the 4-bit/6 GB hardware regime. They are not restated here. Chapter 7 revisits them as interpretive limits once the findings are in hand.

One question can only be assessed after the fact: how much statistical weight these results support. The answer is less than the number of reported digits suggests.

Run-to-run instability dominates the between-model signal. The between-model IFR differences—at most about four points on the mean—are smaller than the within-model swings between repetitions. The clearest case is `codellama-7b`, whose IFR ranges from 0.38 to 0.68 across its three runs, a spread seven times the largest difference between model means. A ranking drawn from these means would be a ranking of sampling noise.

Three repetitions bound what the intervals can show. The hierarchical bootstrap intervals (Table 6.13) overlap almost completely, and the corrected pairwise comparisons are non-significant with negligible effect sizes (all $|\delta| \leq 0.08$, all $p_{\text{Bonferroni}} \approx 1.0$). The repetition count, rather than the analysis, sets the width of those intervals: three runs per model expose gross instability but do not estimate the mean closely, and resampling cannot manufacture precision the design never collected. With a generation process that is non-deterministic by construction—sampling at $\theta = 0.1$ rather than greedy decoding—this is the binding constraint on every inferential claim in this chapter. Ten or more repetitions per model would narrow the intervals enough to make a genuine four-point difference detectable. The design lesson, which is worth separating from the result, is that a study of a stochastic repair process must budget for repetitions and not for corpus size alone.

Consequently, no model ranking is claimed. The absence of a detectable difference must be read as inconclusive, not as demonstrated equivalence: a larger, category-stratified corpus with more repetitions could still reveal a real but small gap. The one comparison that *is* statistically robust is the prompting-strategy effect from the ablation (Section 6.4), where the differences exceed the run-to-run noise by a wide margin—though, as noted there, on a single model.

6.13 Summary of Findings

Table 6.15 consolidates model performance across all dimensions analysed in this chapter.

The results admit a concise practical reading. On repair quality the three models are a statistical tie (Section 6.11): no model can be recommended over another on fix rate, and scale alone, across the 3B–7B range tested, buys no measurable repair quality. The decision therefore turns on cost and risk. `qwen2.5-coder-3b` is the right default for interactive use, IDE plugins and pre-commit hooks, where its third-of-a-second latency matters and its run-to-run stability is good. `codellama-7b` suits unattended batch repair where wall-clock time is free: it is the slowest and least stable model, but it posts the lowest regression rate and the leanest token cost, and in

Table 6.15: Consolidated model comparison (cross-repetition means). Arrows indicate direction of preference: ↑ higher is better, ↓ lower is better. **Bold** indicates best value on each metric. CL = `codellama-7b`; Q3b = `qwen2.5-coder-3b`; Q7b = `qwen2.5-coder-7b`.

Metric	**CL-7b**	**Q3b**	**Q7b**	**Best / Key observation**
IFR ↑	0.504	0.526	**0.544**	Statistically tied; all < 0.70
Complete-rep. SR ↑	**59.6%**	50.5%	54.5%	CL leads but high variance
ρ (H5 <5%) ↓	**5.2%**	11.9%	7.6%	All exceed 5%; CL lowest
MTTR (s) ↓	5.73	**0.33**	0.99	Q3b ≈17× faster than CL
Tokens/fix ↓	**954**	1,326	1,260	CL leanest per fix
IFR stability (SD) ↓	0.155	0.074	**0.069**	CL least stable (timeouts)
H3 (IFR ≥0.70)	×	×	×	Not met by any model

batch mode neither its latency nor its variance is disqualifying. `qwen2.5-coder-7b` sits between the two and is a reasonable middle option, but it has no dimension on which it is clearly the best choice. The largest effect observed anywhere in this study, larger than any model difference, is the prompting strategy: the ablation shows zero-shot roughly quadrupling the fix rate of few-shot (Section 6.4). That contrast, however, was run on `qwen2.5-coder-3b` alone, so it establishes the size of the effect on one model rather than a general ordering of strategies; the reasons to expect it to be capacity-dependent are set out in Section 6.4.

None of the three models reaches the IFR target of 0.70, and the regression rate exceeds the 5% ceiling for all three. Neither result is a failure of the pipeline; both are informative. The IFR shortfall is concentrated in semantic-structure violations, the largest category and the one least amenable to repair under the conditions tested here. Whether that reflects a property of the repair problem, of the information available to the pipeline, or of an oracle that cannot assess meaning is not separable within this design; the three readings are weighed in Section 7.2.2. The regression rate, now measured directly rather than assumed zero, shows that a non-trivial share of plausible-looking patches would break a component's interface, and that Layer 2 of the validation pipeline catches those interface regressions before they reach disk.

The architectural ablation (Section 6.10) adds a more sober note to that picture and answers RQ1. Switching off each structural component in turn does not significantly change the fix rate: at the file level, where observations are actually

independent, the study is underpowered and the effects are null. The honest synthesis across H1 and H5 is therefore a *divided verdict on the pipeline's own machinery*: the functional gate (Layer 2) demonstrably catches interface-breaking patches and earns its keep, while the broader claim that every component lifts the fix rate is not borne out by the data. One result deliberately left open is whether bypassing the validation stack genuinely repairs more semantic violations or merely lets the model delete the offending structure to silence the scanner; the present oracle cannot tell those apart, and settling it is named as direct future work. The failure taxonomy (Table 6.10) separates the engineering-addressable modes (no-valid-code outputs, timeouts) from the epistemic boundary (semantic repairs requiring information the source file cannot provide). Chapter 7 develops this distinction, interprets the results against each research question, and draws out the implications for model selection, prompting strategy, and the limits of the current automation ceiling.

Chapter 7

Discussion

This chapter reads the results against the four research questions (Section 7.1), traces the failure modes that bound the framework's performance (Section 7.2), and draws out what they mean for deployment and the surrounding literature (Section 7.3). It closes with an account of the study's limitations (Section 7.4) and the broader implications of the architecture for other quality domains (Section 7.5).

7.1 Interpreting the Results

The clearest signal in the whole study came from the prompt rather than from the model. RQ2 asked whether the prompting strategy affects repair quality, and it does, by a wide margin: on a fixed model and corpus (Section 6.4), zero-shot prompting resolved 56% of violations against 14% for few-shot and 10% for chain-of-thought (Kruskal–Wallis $p < 0.001$; zero-shot vs. few-shot Cliff's $\delta = -0.34$, medium; vs. chain-of-thought $\delta = 0.47$, large). The same ordering holds on cost (roughly 1,100 output tokens per resolved violation against about 10,700 and 136,000), so the leanest strategy is also the most economical, which is exactly why zero-shot was carried into the canonical experiment. The direction looks counter-intuitive only until one remembers the setting: a 3B model under a 6 GB budget, where retrieved exemplars and a reasoning preamble crowd the context and pull attention away from the one thing that matters, the violation in its surrounding code, while chain-of-thought's long free-form generations drift into edits the file never needed (its 0.34 regression rate is more than triple zero-shot's). For a short, sharply bounded edit, less prompt is more. A companion axis varied the repair scaffolding (a direct call, a surgical patch agent, a broad-context agent) with the strategy fixed, and no scaffold beat the baseline (McNemar $p > 0.05$); that null is bounded, though, because the agent conditions fell back to a prompt-level operationalisation when their command-line tools proved absent, so the axis compares prompt shapes, not live agents with execution feedback.

RQ1 turns the same scrutiny back on the framework itself. Switching off the reflection loop, the few-shot exemplars, the WCAG semantic context, or the internal validation stack, one at a time, leaves the per-file fix rate statistically unchanged (Section 6.10): no comparison survives Holm–Bonferroni, and the point estimates scatter within about two points of the full pipeline in both directions. Given the study's power, the honest reading is that no single component demonstrably lifts repair quality. The result also carries a methodological lesson. An earlier, naive analysis that pooled every violation across every repetition made the components look decisive, with the fix rate apparently climbing from 59% to 89% as they were removed. But violations in one file share a single patch and are not independent, and once the unit of analysis is correctly set to the file, that effect dissolves into sixty-odd ties and a null. The same discipline that replaced the model's claim of success with a browser re-scan applies to the unit of analysis: both corrections are about measuring the right thing at the right grain. Read together, RQ1 and RQ2 point the same way. Quality moves with *how* the model is prompted, and barely at all with *how much* machinery wraps the prompt; removing the few-shot exemplars even cut the token bill by about a third at no measurable cost in fixes. Simplicity competes. One component did look substantial on removal—internal validation—and it is the one component whose effect the fix-rate oracle cannot adjudicate, since the oracle cannot tell a genuine semantic repair from a deletion that silences the scanner. That thread is picked up under Limitations.

Section 4.3 presents qualitative observations attributing specific failure modes to the absence of each template component: removing Component 2 produces semantically incorrect repairs, removing Component 4 increases API hallucination, and so on. The architectural ablation finds none of these effects—no component's removal produces a statistically detectable change in the fix rate at the file level. This is not a methodological coincidence; it is a real contradiction between qualitative inspection on a small convenience sample and a formal test on a larger paired corpus. Two factors resolve most of it. First, the conditions differ: the qualitative observations were made on approximately 30 violations examined visually before any formal evaluation design existed, using a subjective correctness criterion, whereas the formal ablation uses 66 paired files under a strict browser re-scan oracle. Second, the formal test is underpowered for per-component effects: approximately 60 of the 66 file pairs tied, leaving 3–5 effective non-tied pairs per component, a sample too thin to distinguish real signal from noise. The most defensible reading is that individual component effects *may* exist but are too small for the formal test to detect—the qualitative signals were real but weak. Conflating the two levels of evidence would overstate confidence in either direction.

The null raises a harder question than the chapter has so far asked: if no com-

ponent of the architecture demonstrably raises the fix rate, what is the architecture contributing to the roughly half of violations that do get repaired? Three considerations bear on the answer.

The first concerns what the ablation actually measured. It removed components *one at a time* from an otherwise complete pipeline, which yields each component's marginal contribution within a system that still contains all the others. The contribution of the architecture as a whole was never measured: no condition compares the full pipeline against a bare model call with no template, no routing, no validation, and no retry. That comparison is the one that would attribute the aggregate result, and it was never run. Its absence is a genuine gap in the evaluation design, and Section 8.4 records it as one.

The second is that a partially redundant design produces marginal nulls as a matter of course. Several components address overlapping failure modes: the constraint list and the output schema both suppress malformed patches, and the code context and the routing tier both supply surrounding structure. Removing any one leaves the others to absorb its function. A set of individually dispensable components can still be jointly necessary, and the single-removal design cannot distinguish that case from genuine irrelevance; cumulative removal would be required.

The third concerns a contribution the fix rate cannot register, and it is the one with the clearest evidence behind it. The validation stack was built to keep regressive patches out of the codebase, and it did so (Section 7.5). A component that holds the fix rate constant while removing a class of harmful outputs is working as designed, and an IFR-based ablation has no way to credit it. What the evidence supports, then, is a more modest claim than the one originally advertised: the architecture appears not to drive repair quality, but it does make an unreliable generator safe enough to deploy.

Where RQ2 found a large effect, RQ3 found none. The three models post mean IFRs of 0.50, 0.53, and 0.54 (CodeLlama-7B, Qwen2.5-Coder-3B, Qwen2.5-Coder-7B), with every pairwise comparison non-significant after correction (Bonferroni $p = 1.0$, $\delta \approx 0$). At 168 violations the study cannot resolve a few-point gap, so this is equivalence-within-noise rather than a ranking, but the negative result is itself informative: the 7B models are no better than the 3B, so scaling the model up bought no repair quality in this regime, a caution against the reflex of defaulting to the largest model that fits in VRAM. Where the models do separate is cost. Mean time per file spans more than an order of magnitude: 0.33 s for Qwen 3B, 0.99 s for Qwen 7B, 5.73 s for CodeLlama, which additionally incurred 27 generation timeouts and the widest run-to-run spread (SD 0.16 against 0.07 for both Qwen models). Token cost per fix is similar across models and uncorrelated with the indistinguishable fix rates. Because quality is tied, these secondary axes, not the fix

rate, are what should drive model selection (Section 7.3).

RQ4 asks which violations yield, and the answer is a clean split along the structural/semantic line. The corpus is not dominated by any single category (Table 6.3): semantic-structure violations (WCAG 1.3.1) lead at a third of the corpus, ahead of ARIA attributes and a tail of contrast, label, alternative-text, and target-size issues. Single-attribute repairs on a known element resolve well: the well-populated structural types fall in the 55–89% IFR band, with contrast the one model-dependent exception, repaired well by the Qwen models but poorly by CodeLlama. Repairs that require meaning the file does not contain do not: semantic-structure violations, despite being the largest category, resolve at only 14–35%, and the lone keyboard case at zero. By WCAG principle the same split appears, Robust the best-served and Perceivable dragged down by its semantic constituent. This locates the automation ceiling, and its character is epistemic rather than architectural: whether an image is decorative, what a link's destination conveys, or how a heading hierarchy should be reorganised depends on design intent that lives outside the JSX file, and no prompt can supply information that is absent from the context the pipeline assembles. What a differently built pipeline might obtain is a separate question, developed in Section 7.2.2. The smallest cells (keyboard, n=1; alternative text, n=7) are descriptive only, so the H4 stated a priori is reported descriptively (Section 6.9); testing it with power would require a category-stratified corpus.

A tension in the evidence on semantic-structure repairs. The claim that the semantic-structure ceiling is *epistemic* (the model lacks information that no prompt can supply) sits in tension with one result from the architectural ablation. Removing the WCAG semantic-context component (`arch_no_wcag_guidelines`) does not lower the aggregate IFR relative to the full pipeline — but the per-category breakdown shows that on the semantic-structure category specifically, the *no-WCAG-context* condition observes a markedly higher IFR (approximately 0.65 versus 0.16 under the full pipeline on `qwen2.5-coder-7b`). If supplying rich WCAG semantic context *raises* the bar the model is trying to meet — causing it to attempt more ambitious tag restructurings that then fail the oracle's strict re-scan — then the ceiling may be partially *calibration-driven*: adding WCAG context increases the sophistication of attempts at the cost of increasing the failure rate on attempts. Stripping that context may lead to simpler, shallower edits that happen to silence the scanner without satisfying the spirit of the criterion. Both readings deserve caution: the per-category cell in the ablation contains only the semantic-structure violations in the 66-file corpus, giving very limited statistical power, and the oracle (Pa11y re-scan) cannot distinguish a genuine semantic repair from a tag substitution that merely stops the scanner. The result does not overturn the epistemic interpreta-

tion, but it weakens the claim that adding more semantic context is unambiguously beneficial for this violation type; it is at least as consistent with the hypothesis that simpler prompts produce shallower but scanner-passing edits more reliably. Disentangling these requires a manual inspection study or an assistive-technology validation protocol, beyond this dissertation's scope.

7.2 Failure Modes and the Automation Ceiling

Unresolved repairs surface under one dominant message, `"SWE: no valid code or patches found in response"`, which on inspection subsumes two structurally distinct root causes. Table 7.1 places these alongside three further modes seen during development, each with its structural cause, the prompt component meant to mitigate it, and the validation gate that catches residual instances.

The two residual modes define the practical ceiling, for different reasons. *Invalid patch synthesis* concentrates on violations needing context beyond the file under repair: ARIA relationships spanning components, focus traps defined in a parent's lifecycle, link text sourced from a sibling or a data layer. The broad-context agent exists for exactly these cases, but it is invoked on only a small minority of files and cannot shift the aggregate on its own, so widening that routing threshold is the highest-leverage change available without swapping the model. *Superficial addressing* runs deeper. When a model emits `alt="decorative image"` instead of `alt=""`, it has produced a well-formed string that general pre-training treats as plausible and the standard rejects; the domain gate catches it and forces a retry, but a second superficial attempt leaves the violation pending. Closing this mode needs either accessibility-specific fine-tuning or a richer, better-targeted exemplar library—the same few-shot mechanism the strategy ablation showed to be counter-productive as a blanket default. That its value should be conditional and per-category, rather than global, follows directly from the two results read together. The remaining three modes are effectively suppressed: hallucinated attributes and over-engineering are rejected at the syntactic and code-quality gates, and interface-altering edits at the structural-diff gate. The rejection itself matters less than the loop it enables: a rejected patch returns to the agent as structured feedback, turning one-shot generation into a bounded, diagnostically guided retry, a point developed in Section 7.5.

7.2.1 What Happened to Hallucination

Hallucination deserves separate treatment, because the taxonomy splits it in a way that turns out to be the central result of this chapter in miniature.

Section 2.4.4 distinguished two forms. The lexical form invents vocabulary:

Table 7.1: Failure mode taxonomy: root causes, structural location, and prompt mitigations. "Residual" means the failure mode persists despite the mitigation being active in the current framework.

Failure Mode	**Structural Cause**	**Prompt Mitigation**	**Validation Gate**	**Residual**
Invalid patch synthesis	Multi-file coordination; model cannot see dependency context	Component 3 (code context); routing to OpenHands	Layer 1 (syntactic)	Yes
Superficial addressing	Incomplete semantic encoding in pre-training; general corpora encode `alt="decorative image"` more often than `alt=""`	Component 2 (violation context with user-impact framing)	Layer 3 (domain compliance)	Yes
API hallucination	Model invents non-existent ARIA attributes from spurious associations (`aria-role="navigation-button"`)	Component 4 (constraints); Component 6 (JSON schema)	Layer 1 (syntactic) / Layer 3	Rare
Over-engineering	Model favours comprehensive solutions; a colour contrast violation elicits a custom theme hook	Component 4 (`[NO] add unnecessary abstraction`)	Layer 4 (code quality, complexity Δ)	Rare
Context blindness	Fix correct in isolation but inconsistent with codebase (imperative DOM in a declarative project)	Component 3 (styling approach, CSS Modules flag)	Layer 2 (AST interface diff)	Rare

`aria-role="navigation-button"`, an attribute that obeys every surface pattern of ARIA and exists nowhere in the specification. The semantic form uses real vocabulary wrongly: `alt="decorative image"` on a decorative image, where every token is legitimate and the result still defeats the attribute's purpose.

The two fare very differently here, and the reason is not that one is rarer. Lexical hallucination is **decidable against an external reference**. The ARIA specification is a finite list; an attribute is on it or it is not, and a lookup settles the question without judgement. The domain-compliance layer performs exactly that lookup, which is why this mode is suppressed to the point of being marked *rare* in Table 7.1 despite being the first failure the framework was built to catch. Constraining generation did not eliminate it; verifying output did.

Semantic hallucination is **not decidable against any reference the framework has**. Determining that `alt="decorative image"` is wrong requires knowing that the image is decorative, which requires knowing what the image is for—the same missing information that defines the ceiling of Section 7.2.2. A scanner confirms an `alt` attribute is present and stops there. So the patch passes every gate, is credited by the oracle, and enters the fix rate as a success while leaving the user no better served than before.

This has a consequence for how the reported IFR should be read that is sharper than the general oracle-circularity caveat. Some fraction of the credited repairs are semantic hallucinations that survived because nothing in the measurement apparatus can recognise them, and that fraction cannot be estimated from the data collected here—it is one of the quantities the human annotation pass in Section 8.4 exists to obtain. The honest summary is that the framework has solved the hallucination problem it can verify and has not addressed the one it cannot, and that the second is the harder of the two precisely because its outputs are indistinguishable from success.

7.2.2 What Kind of Ceiling Is This?

The residual modes could be read as evidence of a *semantic automation ceiling*: a boundary where accessibility repair stops being a coding problem and becomes an act of interpretation that only a person can perform. That reading goes further than the evidence supports. How much further is worth setting out, since the two claims require different amounts of evidence.

What the experiments establish is narrow and solid: within this framework, on this corpus, under a scanner oracle, using three quantised open-weight models of 3B–7B parameters prompted from source text alone, violations classified as semantic were repaired at a markedly lower rate than structural ones, and no configuration

examined closed that gap. That is a statement about a *measured* ceiling—the observed limit of one system under one set of conditions.

The stronger claim, that the residue is irreducibly human, would require showing that no feasible system could close the gap. Nothing here shows that, and at least four levers were held fixed throughout, each of which plausibly moves the boundary:

Execution context. The models saw source text, never a running interface. Whether a focus order is sensible or a live region announces at the right moment is a property of behaviour over time, and the repair loop was never given access to it.

Visual and multimodal input. Judging whether an image is decorative or informative is close to trivial when the image is visible and close to guesswork from a file path. A vision-capable model receiving the rendered component operates on evidence this study withheld.

Requirements and design intent. Whether a link label is meaningful out of context depends on what the product intends the link to do. That intent lives in issue trackers, design specifications, and product documentation—artefacts outside the repair context window here.

Human feedback in the loop. The framework's retry loop is closed by a scanner. A loop closed by a developer or an assistive-technology user would supply exactly the semantic signal whose absence characterises these failures, converting an unanswerable question into an answerable one.

All four levers are ordinary engineering interventions, and each remains untried. This dissertation therefore locates a ceiling and characterises what sits beneath it, without establishing whether its height is a property of the problem or of the apparatus used to measure it. The formulation the evidence supports is that semantic violations mark the current *frontier* of context-free automated accessibility repair: the point at which returns from better prompting diminish and further progress depends on supplying new kinds of input. Whether an irreducible core survives all four interventions is an open empirical question, and the experiments that would answer it are set out in Section 8.4.

There is also a measurement reason for caution, developed in Section 7.4: the oracle that classifies these violations as unrepaired is itself incapable of semantic judgement, and no human ever audited the patches it rejected. A ceiling identified by an instrument that cannot see past it should be reported as the limit of the instrument until an independent measurement says otherwise.

7.3 Deployment and Positioning

What "human in the loop" means here. The phrase recurs below and in Section 8.4, and it is worth fixing its meaning, because the framework as evaluated is deliberately *not* human-in-the-loop: every result in Chapter 6 was produced by a closed automated cycle in which a scanner both raises the violation and decides whether the patch resolved it, with no person consulted at any point. A human-in-the-loop configuration would insert a person at one of three distinct positions, and the three are not interchangeable. At the *input*, a reviewer confirms or rejects a flagged violation before it enters the repair queue, which addresses false positives. At the *output*, a reviewer accepts or rejects a generated patch before it is committed, which addresses scan-evasion and semantic error. And at the *loop*, the reviewer's rejection replaces the scanner as the retry signal, which is the only one of the three that changes what the model can learn within an attempt budget, since it supplies the semantic feedback the scanner cannot. Where this dissertation proposes human involvement it means the second and third positions; the first is properly a corpus-construction concern and is treated as such in Section 5.7.

Because the three models are statistically tied on quality, the deployment choice turns on latency, stability, and failure handling. Figure 6.2 makes the separation visual: a flat quality axis against a latency axis spanning an order of magnitude. For interactive use (IDE plugins, pre-commit hooks), the binding constraint is sub-two-second response, and Qwen2.5-Coder-3B, at a third of a second per file even with a retry, is the natural fit, escalating failures to a human queue instead of blocking the commit. For unattended batch use latency is slack and consistency matters more; Qwen 3B remains attractive on throughput, while CodeLlama's traditional appeal, stability, is undercut here by its timeouts and high IFR variance, so it is no longer an obvious batch default.

Two further consequences follow from the corpus itself. Every violation was reported by a single tool, which leaves the structural/semantic split as the practical escalation signal: route semantic-structure and link-purpose violations to human review by default. And roughly half of all confirmed violations survive three repair attempts, so any deployment must present automation as a first pass over a backlog and never as a closed-loop fix-all.

These choices also locate the work against the literature, where the absence of shared benchmarks limits direct numerical comparison but three contrasts place it. Against prompt-engineering-only approaches, Huang et al. [76] report 51% violation reduction applying GPT-3.5 to static HTML without fine-tuning; the present framework reaches a comparable band (IFR 0.50–0.54) on production React/TypeScript. This is not a head-to-head benchmark (a proprietary model orders of magnitude

larger, a simpler input medium, and a possibly different metric all preclude that), but it is non-trivial that small open-weight models running locally land in roughly the same range as a large proprietary API on a harder medium, without fine-tuning. Against supervised fine-tuning, embedding accessibility knowledge in weights can in principle raise IFR, at the cost of curating a labelled corpus and re-training as taxonomies evolve (the WCAG 2.2 additions, say); editing a violation-context database and an exemplar library is the cheaper, more maintainable alternative, and the prompt-only baseline is competitive enough to make that a genuine decision rather than a foregone one. Against general-purpose coding agents, SWE-agent [15] and OpenHands [16] resolve real GitHub issues at rates that scale with model capability; the present work suggests a structured, domain-specific prompt is enough to carry such agents into accessibility without fine-tuning, though, as the scaffolding ablation noted, the agent CLIs were not exercised end-to-end, so this is a claim about domain transfer of the prompt design rather than a measured comparison of runtimes. Underlying all three contrasts, the framework is the engineering realisation of the Rapid Review (Chapter 3): structured prompts with explicit constraints and output contracts, retrieval-based grounding, and composite validation oracles were identified there as first-order drivers of repair quality across the included studies, and the six-component template plus four-layer pipeline operationalises that prescription directly.

7.4 Limitations

Several limitations bound the conclusions, and most reduce to the same root: the scarcity of confirmed real-world violations. The model comparison is underpowered: with 168 violations in 33 files it cannot resolve IFR differences of a few points, so the models' indistinguishability is an absence of detectable difference, not demonstrated equivalence, and a larger, category-stratified corpus is the single most valuable improvement to the evidence base. The strategy result, in turn, generalises from one model: the ablation that selected zero-shot ran on the 3B alone, and the mechanisms proposed for its advantage, context dilution and reasoning-induced instability, are plausibly weaker for larger models with more headroom, so zero-shot's superiority should not be assumed universal; re-running the strategy axis per model is a direct extension, and the scaffolding axis, having compared prompt shapes rather than live agents, bounds what can be said about agentic tooling.

The architectural ablation shares those limits and adds one of its own. It too ran on a single model and is underpowered at the file level, so its null on component contribution is again an absence of detectable effect rather than proof of irrelevance. A functional-preservation check now guards against the obvious artefact: applied

to the four gated conditions, it found no fix that resolved a violation by removing an export, prop, handler, heading, landmark, or content, so their fix rates are functionally clean. The one unresolved point is the internal-validation condition, which by design switches that check off: bypassing the validation stack raised the *scanner-measured* fix rate, but the gain sits almost entirely in semantic structure, the one category where the cheapest way to satisfy the scanner is to delete the offending element rather than repair it. For that condition the fix-rate oracle cannot tell a genuine structural repair from such a deletion, so it cannot, on its own, decide whether the validation stack was over-rejecting good patches or correctly intercepting destructive ones; running that condition's patches through the functional check would settle the matter, and is named as future work (Section 8.4).

The deepest limitation is the oracle itself. A fix is credited when a browser re-scan no longer reports the violation, and that re-scan shares the rule implementations used for detection, so scanner compliance is a necessary but not sufficient proxy for accessibility as experienced through assistive technology; a companion construct-validity audit indicates that a minority of credited fixes are superficial or address scanner false positives, and no screen-reader or keyboard-user study was conducted. The reported IFR should therefore be read as scanner-verified repair, an upper bound on genuine improvement. Finally, scope and hardware bound external validity: the corpus is React/TypeScript only, and all models ran at 4-bit quantisation on 6 GB of VRAM, so the absence of a model-size advantage holds within that regime and may not survive full-precision or larger-VRAM inference, where the 7B models could recover headroom (Section 5.7).

7.5 Implications

Beyond accessibility, the work carries a few implications. The first is about the detection-to-correction gap that motivated it: noncompliance persists because detection and correction are handled as separate, human-mediated steps, and this work shows that for any domain with structured detection outputs, a programmatic validation oracle, and a documented repair knowledge base, the correction step can be automated by a systematic detection-to-repair transformation without fine-tuning. Security linting, code style, and API migration all meet those conditions and could be targeted by re-configuring the violation-context and domain-compliance layers.

How much of this is React-specific? The evaluation ran exclusively on React/TypeScript, so the numbers do not transfer. The *design*, however, is React-specific in fewer places than the corpus suggests, and it is worth separating the two, because the cost of retargeting the framework depends entirely on which layers

move.

Three components carry no framework assumption at all. The six-component prompt template is a schema over a violation report, a code excerpt, and a set of constraints; nothing in its structure names a framework. The adaptive router scores a file by violation count, type diversity, and complexity, none of which is React-derived. And the bounded validate-and-retry loop, which Section 7.5 identifies as the framework's primary architectural contribution, is indifferent to what is being repaired.

Two components are genuinely coupled. Layer 2 extracts prop interfaces, exports, and event-handler bindings through a TypeScript AST, so retargeting to Vue or Angular means replacing the parser and redefining what "interface preservation" denotes in that component model—a bounded engineering task, not a redesign. The harness is the harder case: it mounts a component in isolation so that browser-based scanners have something to scan, and that mounting procedure is per-framework. Frameworks with weaker isolation stories, or server-rendered stacks where a component has no standalone rendering, would need a different detection strategy rather than a different mounting call.

The honest summary is that the contribution generalises at the level of *architecture* and not at the level of *instrument*: another framework inherits the template, the routing, and the retry loop, and must supply its own AST adapter and rendering harness. Whether the repair rates survive that transfer is untested and is not claimed.

That transfer claim raises a question worth answering explicitly: would the semantic ceiling observed here reappear in those domains? The mechanism identified in Section 7.2.2 suggests it would, unevenly, and with a severity that can be anticipated. Violations resisted repair here when deciding on the correct edit required information the codebase does not contain—design intent, user context, the purpose a component serves. Applying the edit was never the difficulty. The ceiling should therefore track how much of a domain's correctness condition is fixed by the artefact itself.

Code style sits at the favourable extreme: the standard is fully specified and mechanically checkable, the artefact contains everything needed to decide conformance, and there is little for a ceiling to consist of. API migration sits in the middle, mostly determined by the old and new interface contracts, with a residue of cases where the right migration depends on why the original call was written the way it was. Security is the closest analogue to accessibility and probably the least favourable: whether a value is attacker-controlled, whether a permission check is appropriate for a given endpoint, or whether data is sensitive are judgements about the system's threat model and deployment context, none of which is recoverable from the file un-

der repair. A security-repair study built on this framework should expect the same structural/semantic split, with the semantic side populated by context-dependent authorisation and trust-boundary issues.

The general prediction is that domains whose oracles are complete admit high automation, while domains whose oracles are proxies for a human judgement—as accessibility scanners are—will show a ceiling roughly where the proxy stops tracking the judgement. Extrapolating from a single domain, this is a testable hypothesis and is offered as a direction for further work rather than as a finding of this dissertation.

The second implication concerns model selection, and it is a negative finding with practical teeth: at $\leq$7B and 4-bit, larger did not mean better, and the prompting strategy moved the fix rate where the architecture did not. Practitioners should benchmark candidate models on their own task rather than default to size, and should treat the prompt configuration as the first thing to tune. The third reframes the role of validation. Discarding bad patches is the smaller part of its value; the larger part is that each rejection returns to the agent as structured feedback, recasting repair from one-shot generation into a bounded, guided search—a validate-and-retry pattern transferable to any repair domain and the framework's primary architectural contribution. It also prompts a precise reading of the regression result, because the H5 metric and the pipeline's operational outcome point in different directions and must be kept separate.

(a) H5 metric (descriptive; exceeds 5% reference). The functional regression rate ρ measures the fraction of accepted-and-applied patches that altered the component's prop interface, exports, or event handlers. Across the three models this rate is 5.2% (CodeLlama), 7.6% (Qwen 7B), and 11.9% (Qwen 3B), all above the 5% practical reference point for H5. As noted in Section 5.2, H5 is a descriptive metric: no null hypothesis was pre-specified and no formal rejection is claimed; the 5% figure is a practical reference, not a threshold derived from prior work or power analysis.

(b) Operational implication (positive). The patches counted in ρ were *proposed* but never committed. The validation pipeline's Layer 2 functional-preservation gate flagged and discarded every regressive patch before it touched the repository. No regressive modification reached a developer's codebase: the commits produced by the framework are functionally clean by the Layer 2 criterion. In other words, H5 characterises what the model attempted, while the pipeline characterises what actually reached code. Both facts matter and neither supersedes the other: H5 signals that the model generation is not regression-free on its own; the pipeline result signals that the validation layer successfully contains that risk.

Finally, the regulatory setting that motivated the work sharpens its relevance. The *Lei Brasileira de Inclusão* (Law No. 13.146/2015) mandates WCAG-conformant

accessibility for companies and public bodies in Brazil, and the 2025 standard ABNT NBR 17225 operationalises it with concrete criteria [8, 10]. The violations the framework handles best (missing ARIA names, absent label associations, contrast) are exactly the mechanical compliance failures such audits surface, and automating roughly half of them, with regressive patches blocked before commit, translates into reduced and auditable compliance risk, leaving human reviewers for the semantic judgements that neither the standard nor the framework can resolve without stakeholder input.

Chapter 8

Conclusion

8.1 Summary of the Work

Web accessibility has long been characterised by a structural asymmetry: detection is automated and inexpensive; correction is manual and costly. This noncompliance persists not because developers lack awareness of violations: mature tools such as Pa11y, Axe-core, and ESLint-jsx-a11y surface them at zero cost. It persists because transforming a structured diagnostic report into a valid, tested, production-ready code fix requires a chain of contextual judgements that no detection tool provides.

This dissertation addressed that gap with *a11y-autofix*, a four-phase automated repair framework covering the full detection-to-correction pipeline for WCAG 2.1 AA violations in React/TypeScript codebases. The framework was designed around a central insight: the information needed for repair is largely already present in existing tooling. The violation report identifies the criterion and the element. The CSS selector maps to a JSX source location via Babel AST traversal. The WCAG normative specification provides the full semantic intent. The component source supplies the architectural constraints. What was missing was a *systematic transformation*: a structured procedure that extracts, enriches, and packages this information into a prompt that reliably guides an LLM toward a correct, minimal, framework-appropriate patch. That transformation is the six-component prompt template.

The framework was evaluated in three stages on a benchmark of 1,249 React/TypeScript files drawn from 36 open-source GitHub projects, stratified across seven application domains, three project-size tiers, and three popularity tiers, containing 168 confirmed WCAG 2.1 AA violations across 33 files and balanced across seven issue types. A first stage, an ablation on a single model, isolated the prompting strategy and selected zero-shot as the strongest option. A second stage, the canonical experiment, then compared three open-weight code-generation models under a

6 GB VRAM constraint across three stability-check repetitions (seed fixed at 42, temperature at 0.1; see Section 6.3), with every repair credited by a browser re-scan of the patched file. A third stage, the architectural ablation, switched off each structural component in turn and found none of them, on its own, to move the fix rate by a detectable margin. The result is a reproducible performance characterisation in which, within the range of configurations examined here, the prompting strategy accounted for larger differences in repair quality than either the choice of model or the accumulation of pipeline components.

8.2 Summary of Research Questions, Contributions, and Hypotheses

Table 8.1 consolidates the answers to the four research questions, the four primary contributions, and the five a-priori hypothesis outcomes in a single reference. Sections 6.11 and 7 provide the supporting evidence and detailed interpretation for each entry.

Table 8.1: Synthesis of research question answers, contributions, and hypothesis outcomes. CL = `codellama-7b`; Q3b = `qwen2.5-coder-3b`; Q7b = `qwen2.5-coder-7b`.

Item	Result / Outcome	Primary implication
RQ1 (component ablation)	**Answered** (single model, 7B). Removing the reflection loop, few-shot exemplars, WCAG context, or internal validation, one at a time, leaves the per-file fix rate statistically unchanged (no comparison survives Holm–Bonferroni; n=66 files, underpowered).	No single component demonstrably lifts the fix rate; the prompting strategy (RQ2) carries the performance, not the accumulated scaffolding.

Item	Result / Outcome	Primary implication
RQ2 (strategies)	**Answered** (single model). Zero-shot far outperforms few-shot (≈56% vs 14% IFR) and chain-of-thought (≈10%); large, significant effects ($p \approx 0.0003$). It also dominates on token cost (≈1.1k tokens/fix vs ≈11k and ≈136k).	On this model, the prompting strategy moved both quality and cost more than any other factor examined; the template alone beats added exemplars or reasoning scaffolds. Measured on `qwen2.5-coder-3b` only; generality across model sizes is untested.
RQ3 (models)	The three models are statistically indistinguishable on IFR (50–54%; pairwise $p \approx 1$, $\lvert\delta\rvert \leq 0.08$). They separate only on cost: Q3b fastest (0.33 s/file), CL-7b slowest, least stable, but lowest ρ and token cost.	Within 3B–7B at 4-bit quantisation, parameter count does not predict repair quality; model choice is a cost/risk decision, not a quality one.
RQ4 (categories)	Attribute-level repairs (ARIA, contrast, alt-text, label, target size): 55–89% IFR. Semantic structure, the largest category (33%), is the residual: 14–35% IFR.	The primary interpretation is that the ceiling is epistemically bounded, not architecturally bounded. However, the architectural ablation shows a counterintuitive result: removing WCAG semantic context raises semantic-structure IFR, possibly because simpler prompts produce shallower edits that still pass the scanner (see Section 7.2). Structural compliance is automatable; the semantic ceiling requires further investigation.
Contribution 1	Six-component prompt template + four-layer validation pipeline with re-scan-verified fix crediting.	Domain-agnostic architecture: reconfigurable for security, style, API-migration domains.

Item	Result / Outcome	Primary implication
Contribution 2	Three-stage evaluation (prompting ablation + 3-model comparison + architectural ablation) on 168 violations, 1,249 files, 3 repetitions; 6 GB VRAM constraint.	Reproducible privacy-preserving baseline for commodity-hardware repair.
Contribution 3	LLM failure-mode taxonomy mapped to prompt components and validation layers.	Separates engineering-addressable failures from the epistemic ceiling.
Contribution 4	Benchmark corpus (36 projects, 1,249 files, 168 violations, balanced across 7 types) + open-source replication package.	Enables independent replication and future model benchmarking.
H2 (IFR by strategy, RQ2)	Confirmed. Zero-shot $\gg$ few-shot, CoT (medium-to-large effects).	Validates the canonical strategy choice and locates the main lever in prompting.
H3 (IFR $\geq$ 0.70)	Not met. Best model Q7b at 0.544; all bootstrap CIs below 0.70.	The gap is informative: it locates the ceiling in semantic repairs, not in engineering failures.
H5 (regression ρ; practical ref. $< 5\%$; descriptive)	Observed 5.2–11.9% (exceeds the 5% reference), but every flagged regression is caught at Layer 2 and never written to disk. No formal null hypothesis was pre-specified; no statistical test was applied (see Section 5.2).	A real fraction of plausible patches would break interfaces; the validation pipeline is what makes autonomous deployment safe.
H1 (component ablation, RQ1)	Not supported: no ablated variant is significantly worse than the full pipeline (architectural ablation, file-level, Holm–Bonferroni).	At this power, no evidence that any single component lifts the fix rate; simplicity competes.
H4 (IFR by category, RQ4)	Descriptive only: two categories have $n \leq 7$.	Per-type rates show semantic repairs as the residual; an inferential test requires a category-stratified benchmark.

8.3 Limitations

Five limitations bound the generalisability of the findings, discussed in full in Sections 5.7 and 7.4. The evaluation is restricted to React/TypeScript projects; as detailed in Section 5.7, the framework's React-specific components would require re-instantiation for other frameworks. As noted in Section 5.1, repair success is measured at scanner-compliance level, which creates an oracle circularity; no assistive-technology user study was conducted. RQ1 was answered by the architectural ablation, but on a single model and at a sample size (66 files) too small to detect a component-level effect, so its null is an absence of detectable contribution rather than proof of irrelevance; and the one component whose removal looked substantial, internal validation, raised a scanner-measured fix rate the oracle cannot distinguish from structural deletion. All results are from 4-bit quantised models on a 6 GB VRAM device, and the effect of quantisation on output quality cannot be disentangled from architectural differences without a full-precision comparison; this is the more salient given that the three models proved statistically indistinguishable on quality.

A fifth limitation concerns statistical power. The corrected corpus is well balanced across issue types, but it contains 168 violations in 33 files, which is enough to characterise aggregate performance yet too few to resolve small between-model differences or to support the inter-category test stated a priori for H4 (two categories have $n \leq 7$). The model comparison should therefore be read as "no detectable difference" rather than "proven equivalence," and constructing a larger, violation-category-stratified corpus is a concrete methodological priority for future work.

8.4 Future Work

Several directions are directly motivated by the limitations and open questions identified in this work, each building on infrastructure that is already implemented or partially in place.

A human-annotated subset of the benchmark is the highest priority, and several of the other directions depend on it. No violation in the corpus and no patch produced against it was ever judged by a person (Section 5.7). Two annotation passes would settle what the scanner cannot. In a validity pass, an accessibility practitioner would confirm or reject each of the 168 violations, converting the corpus from scanner-confirmed to expert-verified and bounding the false-positive rate that is currently only heuristically estimated. In an outcome pass, the same practitioner would judge each credited fix as a genuine improvement, a cosmetic change, or a scan-evasion, yielding the quantity this dissertation could not measure: the gap

between scanner-verified repair and accessibility as experienced through assistive technology. Since the corpus is released as a standalone artefact (Section 5.3.1), these annotations can be layered onto it and would upgrade the benchmark for every subsequent user.

An attribution baseline for the architecture as a whole. The architectural ablation removed components one at a time and found no single one to matter, but no condition compared the complete pipeline against a bare model call—no template, no routing, no validation, no retry (Section 7.1). That comparison would establish what the architecture contributes in aggregate, and it is cheap to run on existing infrastructure. It should be paired with a *cumulative* ablation that strips components in sequence rather than singly, since a design with overlapping mitigations can consist of individually dispensable but jointly necessary parts, and the one-at-a-time design cannot tell that case apart from genuine irrelevance.

Testing whether the semantic ceiling moves. Four inputs were withheld from the repair loop throughout, each a candidate for shifting the boundary identified in Section 7.2.2: the rendered interface supplied to a vision-capable model, runtime execution traces and focus behaviour, product and design intent retrieved from issue trackers or specifications, and a retry loop closed by a developer rather than by a scanner. Reintroducing them one at a time, on the semantic violations that resisted every configuration here, would convert an open question about the ceiling's nature into a measurement of its height.

Strengthening the Rapid Review's evidence base. Three of the review's concessions are directly repairable. Its search rests on a single database with no citation-chasing protocol (Appendix B.6): backward and forward snowballing seeded from the 55 included studies, with the yield enumerated, would bound the coverage bias that is currently only acknowledged, as would replicating the search in the ACM Digital Library and IEEE Xplore. Its synthesis rests on a single coder with no second-researcher review (Section 3.4.1): independent re-coding of the corpus by a second researcher, with agreement reported, would establish whether the four architectural patterns and the five gaps survive a different reading, and the per-study assignment is already published in a form that makes this possible without repeating the search. And no primary study was appraised for methodological rigour, so a quality-assessment pass would let convergence in the corpus be weighted by evidential strength rather than by frequency of mention.

The remaining directions concern the repair engine itself. The most immediate is to settle the one question the architectural ablation leaves open (RQ1). A functional-preservation check, which flags any fix that removes an export, prop, handler, heading, landmark, or visible content, has already been applied to the four gated conditions and found *zero* regressions among their credited fixes; what

remains is to run the internal-validation (bypass) condition's patches through the same check. That bypass raised the scanner-measured fix rate, but the gain was concentrated in semantic structure, where deleting the offending element silences the scanner as cheaply as repairing it, and the fix-rate oracle alone cannot tell the two apart. Splitting that condition's fix rate into genuine repairs and scanner-silencing deletions would convert the ablation's most provocative result from suggestive to decisive. Running the architectural ablation across all three models, rather than the 7B alone, and on a larger corpus would at the same time give it the statistical power its file-level null currently lacks. A second priority follows from the RQ2 result: because the strategy ablation covered only `qwen2.5-coder-3b`, the next step is a model × strategy factorial, since the optimal strategy may well be model-dependent (a larger model with more context headroom might, unlike the 3B model, benefit from few-shot exemplars rather than be diluted by them). Closely related is the construction of a violation-category-stratified corpus with at least 30 instances per type, which would turn the descriptive per-type fix rates of Section 6.9, where semantic-structure repairs already stand out as the residual, into an inferential answer to RQ4.

Two architectural extensions follow naturally from the framework's current design. Cross-run exemplar retrieval would treat each successfully repaired violation as a new exemplar, indexing it by its violation embedding and making it available to subsequent repairs so that the few-shot library grows with each deployment, increasingly grounding the agent in the conventions of the specific codebase under repair. The retrieval infrastructure (cosine similarity over sentence embeddings) is already in place, and extending it to a persistent, updateable index is a straightforward engineering task.

Two further directions concern deployment and human oversight rather than the core repair engine. Integrating the framework into a CI/CD pipeline introduces requirements that the current batch evaluation does not exercise: incremental scanning of changed files, conflict resolution with concurrent branch repairs, and rollback mechanisms for patches that pass automated validation but fail code review. A longitudinal production deployment study would assess whether the validation pipeline's ability to intercept interface-breaking patches, which held across the controlled experiment even as the raw regression rate ran between 5% and 12%, continues to hold under real repository dynamics, including refactors, dependency updates, and framework version migrations that alter the JSX patterns the model has seen. A complementary direction is a human-in-the-loop tier that routes high-confidence violations and all semantic repair attempts to a developer review queue, surfacing the agent's proposed fix alongside the violation context and the WCAG normative description. This hybrid mode would combine the framework's speed ad-

vantage for structural repairs with human judgment for semantic ones, while also collecting human-validated repair examples that enrich the few-shot library over time. Measuring the reduction in developer time-to-repair under this mode, compared to unassisted manual correction, is the most direct measure of the framework's practical value that has not yet been quantified.

The framework's domain-agnostic architecture also makes extension to other quality domains feasible at low cost, as detailed in Section 7.5.

8.5 Closing Statement

This dissertation set out to close one half of a long-standing asymmetry in software quality: detection has been automated for years, correction has not. For web accessibility, and by architectural extension for any quality domain with a formal violation specification, a programmatic validation oracle, and a documented repair knowledge base (Section 7.5), it shows that the correction half is addressable too, through a systematic detection-to-repair transformation, on commodity hardware, with no model fine-tuning or curated training data.

Three lessons outlast the specific numbers. The first is where automation stopped here. About half of the violations resolved reliably under the scanner oracle, while the unresolved half concentrated in semantic-structure repairs, which turn on product judgment—what a link is for, what an image conveys, how a region should read to a screen-reader user—more than on engineering judgment. This dissertation locates that boundary and characterises what lies beneath it, without establishing that the boundary is permanent. The models here worked from source text alone, without a rendered interface, without execution, without design intent, and without a human in the retry loop; every one of those absences is an engineering choice that a future system can reverse (Section 7.2.2). What the study identifies is the frontier of context-free automated repair, which is a narrower claim than an irreducibly human core.

The second concerns where the leverage was. Within the narrow band examined, three quantised open-weight models between 3B and 7B, prompt design mattered more than model choice. Practitioners working under the same constraints can treat the prompt as the first lever and, in this size class, the model as close to a commodity. Whether the ordering survives at larger scales this study did not test.

The third concerns safety, and it carries the most direct engineering consequence. The models proposed regressive patches at a rate between 5% and 12%—patches that would have removed an export, a prop, an event handler, or visible content from a working component. None reached the codebase. The structural interface-preservation gate identified and discarded every one before commit, so that what the

generator *attempted* and what the pipeline *delivered* came apart cleanly: unreliable generation upstream, functionally clean commits downstream. That result reframes what the architecture is for. Its measurable contribution lay in containing a failure mode that would otherwise make an LLM repair tool unsafe to run unattended on production code, and the ablation found no component that raised the fix rate. For automated repair to be adopted in practice, bounding the damage of a wrong answer is worth at least as much as increasing the supply of right ones, and it is the part of this design the evidence most clearly supports.

The information needed for repair was always there, scattered across the scanners, the WCAG specification, and the source itself; what was missing was the infrastructure to extract it, structure it for an agent, and validate the result without a human in the loop. That infrastructure now exists, is documented, and is available for replication and extension. The work has been disseminated as three peer-reviewed publications: the initial repair concept as a workshop paper at the 2nd International Workshop on Human Factors for Software Dependability, co-located with ISSRE 2025 [17]; the full framework and evaluation as a full research paper accepted (early decision) in the Research Track of SBES 2026 [18]; and the measurement methodology underpinning the evaluation at the 1st Workshop on Software Engineering for Agentic Systems, co-located with CBSoft 2026 [19].

References

[1] WORLD HEALTH ORGANIZATION. “Disability and Health”. https://www.who.int/news-room/fact-sheets/detail/disability-and-health, 2023. Accessed: 2024-10-01.

[2] WEBAIM. “The WebAIM Million: Annual Accessibility Analysis”. https://webaim.org/projects/million/, 2024. Accessed: 2024-10-01.

[3] PA11Y. “Pa11y: Automated Accessibility Testing”. https://pa11y.org/, 2024. Accessed: 2024-10-01.

[4] DEQUE SYSTEMS. “axe DevTools: Accessibility Testing for Development Teams”. https://www.deque.com/axe/, 2024. Accessed: 2024-10-01.

[5] GOOGLE. “Lighthouse: Automated Auditing for Web Pages”. https://developer.chrome.com/docs/lighthouse/overview/, 2024. Accessed: 2024-10-01.

[6] IBGE. “Censo Demográfico 2022: Pessoas com Deficiência e Pessoas Diagnosticadas com Transtorno do Espectro Autista — Resultados preliminares da amostra”. https://agenciadenoticias.ibge.gov.br/agencia-noticias/2012-agencia-de-noticias/noticias/43463, 2025. Accessed: 2025-06-01.

[7] IBGE. “Pesquisa Nacional por Amostra de Domicílios (PNAD): Pessoas com Deficiência 2022”. https://www.gov.br/mdh/pt-br/assuntos/noticias/2023/julho/brasil-tem-18-6-milhoes-de-pessoas-com-deficiencia-indica-pesquisa-divulgada-pelo-ibge-e-mdhc, 2023. Accessed: 2024-10-01.

[8] BRAZIL. “Lei nº 13.146, de 6 de julho de 2015: Lei Brasileira de Inclusão da Pessoa com Deficiência (Estatuto da Pessoa com Deficiência)”. https://www.planalto.gov.br/ccivil_03/_ato2015-2018/2015/lei/l13146.htm, 2015. Accessed: 2024-10-01.

[9] EQUALWEB BRASIL. “Lei Brasileira de Inclusão Completa 5 Anos, mas a Acessibilidade Digital Ainda é Algo Distante nas Empresas”. https://equalweb.com.br/lei-brasileira-de-inclusao-completa-5-anos-mas-a-acessibilidade-digital-ainda-e-algo-distante-nas-empresas/, 2021. Accessed: 2024-10-01.

[10] ABNT. “ABNT NBR 17225: Acessibilidade Digital — Requisitos para Sítios e Aplicações Web”. https://www.abnt.org.br, 2025. Accessed: 2025-06-01.

[11] CHEN, M., TWOREK, J., JUN, H., et al. “Evaluating Large Language Models Trained on Code”, *arXiv preprint arXiv:2107.03374*, 2021.

[12] ROZIÈRE, B., GEHRING, J., GLOECKLE, F., et al. “Code Llama: Open Foundation Models for Code”, *arXiv preprint arXiv:2308.12950*, 2023.

[13] QWEN TEAM. “Qwen2.5-Coder: Code-Specific Large Language Model”. arXiv preprint arXiv:2409.12186, 2024.

[14] DEEPSEEK TEAM. “DeepSeek-Coder: When the Large Language Model Meets Programming”. arXiv preprint arXiv:2401.14196, 2024.

[15] YANG, J., JIMENEZ, C. E., WETTIG, A., et al. “SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering”. In: *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. Available at: <https://arxiv.org/abs/2405.15793>.

[16] WANG, X., CHEN, B., YUAN, Y., et al. “OpenHands: An Open Platform for AI Software Developers as Generalist Agents”, *arXiv preprint arXiv:2407.16741*, 2024.

[17] WIELAND, J. P. V. B., WERNER, C. M. L. “Designing for Inclusion: Human-Centered Multi-Agent Accessibility Repair in Modern Web Applications”. In: *Proceedings of the 2nd International Workshop on Human Factors for Software Dependability, co-located with the 36th IEEE International Symposium on Software Reliability Engineering (ISSRE 2025)*. IEEE, 2025.

[18] WIELAND, J. P. V. B., WERNER, C. M. L. “Bridging the Detection-Correction Gap: A Prompt Engineering Framework for Automated Accessibility Repair”. In: *Proceedings of the 40th Brazilian Symposium on Software Engineering (SBES 2026), Research Track*, 2026. Accepted (early decision).

[19] WIELAND, J. P., CATRO, D., WERNER, C. “Measure the Right Thing, at the Right Grain: Three Measurement Traps in Evaluating Code-Repair Agents”. In: *Proceedings of the 1st Workshop on Software Engineering for Agentic Systems (SE4AS), co-located with the Brazilian Conference on Software: Theory and Practice (CBSoft 2026)*, 2026. Accepted.

[20] W3C. *Web Content Accessibility Guidelines (WCAG) 2.1*. Technical report, World Wide Web Consortium, 2018. Available at: <https://www.w3.org/TR/WCAG21/>.

[21] W3C. “ARIA Authoring Practices Guide (APG)”. https://www.w3.org/WAI/ARIA/apg/, 2023. Accessed: 2024-10-01.

[22] MICROSOFT. “Microsoft Playwright Testing”. https://playwright.dev/, 2023.

[23] EVENSON, R. “eslint-plugin-jsx-a11y”. https://github.com/jsx-eslint/eslint-plugin-jsx-a11y, 2024. Accessed: 2024-10-01.

[24] VIGO, M., BROWN, J., CONWAY, V. “Benchmarking Web Accessibility Evaluation Tools: Measuring the Harm of Sole Reliance on Automated Tests”. In: *Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility (W4A ’13)*, pp. 1–10. ACM, 2013. doi: 10.1145/2461121.2461124.

[25] ISMAILOVA, L., INAL, Y. *Comparative Analysis of Automated Accessibility Evaluation Tools*. Technical report, 2022. Available at: <https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3052853>.

[26] LE GOUES, C., NGUYEN, T., FORREST, S., et al. “GenProg: A Generic Method for Automatic Software Repair”, *IEEE Transactions on Software Engineering*, v. 38, n. 1, pp. 54–72, 2012.

[27] LE GOUES, C., OTHERS. “Automated Program Repair”, *Communications of the ACM*, v. 62, n. 12, pp. 56–65, 2019. doi: 10.1145/3338906.

[28] OUYANG, L., WU, J., JIANG, X., et al. “Training Language Models to Follow Instructions with Human Feedback”, *Advances in Neural Information Processing Systems*, v. 35, pp. 27730–27744, 2022.

[29] YAO, S., ZHAO, J., YU, D., et al. “ReAct: Synergizing Reasoning and Acting in Language Models”. In: *Proceedings of the 11th International Conference on Learning Representations (ICLR)*, 2023. Available at: <https://arxiv.org/abs/2210.03629>.

[30] AKILESH, R., OTHERS. “Multi-Agent Hierarchical Workflow for Autonomous Code Generation with Large Language Models”. In: *2025 IEEE International Conference on Nascent Technologies in Engineering (SCEECS)*, 2025. doi: 10.1109/SCEECS64059.2025.10940635.

[31] BOUZENIA, I., DEVANBU, P., PRADEL, M. “RepairAgent: An Autonomous, LLM-Based Agent for Program Repair”. In: *Proceedings of the 47th International Conference on Software Engineering (ICSE)*, 2025. doi: 10.1109/ICSE55347.2025.00157. Available at: <https://arxiv.org/abs/2403.17134>.

[32] ZHANG, J., CHEN, J., LIU, B., et al. “AutoCodeRover: Autonomous Program Improvement”. In: *Proceedings of the ACM on Software Engineering*, 2024. doi: 10.1145/3663529.3663875. Available at: <https://arxiv.org/abs/2404.05427>.

[33] WHITE, J., FU, Q., HAYS, S., et al. “A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT”. arXiv preprint arXiv:2302.11382, 2023.

[34] BROWN, T. B., MANN, B., RYDER, N., et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems (NeurIPS)*, v. 33, pp. 1877–1901, 2020.

[35] MIN, S., LYU, X., HOLTZMAN, A., et al. “Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?” In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 11048–11064. Association for Computational Linguistics, 2022.

[36] WEI, J., WANG, X., SCHUURMANS, D., et al. “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”. In: *Advances in Neural Information Processing Systems (NeurIPS)*, v. 35, pp. 24824–24837, 2022.

[37] SINGH, A., OTHERS. “Enhancing AI Systems with Agentic Workflows Patterns in Large Language Model”. In: *2024 IEEE World Forum on the Internet of Things (WF-IoT)*, 2024. doi: 10.1109/AIIoT61789.2024.10578990.

[38] DING, Z., OTHERS. “Reasoning and Planning with Large Language Models in Code Development”. In: *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2024. doi: 10.1145/3637528.3671452.

[39] MEYES, R., LU, M., DE PUISEAU, C. W., et al. “Ablation Studies in Artificial Neural Networks”, *arXiv preprint arXiv:1901.08644*, 2019. doi: 10.48550/arXiv.1901.08644. Available at: <https://arxiv.org/abs/1901.08644>.

[40] SHEIKHOLESLAMI, S., MEISTER, M., WANG, T., et al. “AutoAblation: Automated Parallel Ablation Studies for Deep Learning”. In: *Proceedings of the 1st Workshop on Machine Learning and Systems (EuroMLSys)*, pp. 55–61, 2021. doi: 10.1145/3437984.3458834.

[41] WOHLIN, C., RUNESON, P., HÖST, M., et al. *Experimentation in Software Engineering*. Springer, 2012. ISBN: 978-3-642-29043-5. doi: 10.1007/978-3-642-29044-2.

[42] STATE OF JS. “State of JavaScript 2023”. https://stateofjs.com/, 2023. Accessed: 2024-06-01.

[43] KITCHENHAM, B., CHARTERS, S. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE-2007-01, Keele University and Durham University Joint Report, 2007.

[44] RAMANAN, K., OTHERS. “ASPIRE: Multi-Agent Execution-Free Program Repair”. In: *Proceedings of the ACM on Software Engineering*, 2024. doi: 10.1145/3663529.3663869.

[45] GUO, Y., OTHERS. “Smart Contract Code Repair Recommendation Based on Reinforcement Learning and Multi-Metric Optimization”, *ACM Transactions on Software Engineering and Methodology*, 2024. doi: 10.1145/3637229.

[46] ZHANG, Y., OTHERS. “Can AI Fix Buggy Code? Exploring the Use of Large Language Models in Automated Program Repair”, *Computer*, 2025. doi: 10.1109/MC.2025.3527407.

[47] BARBARROXA, V., OTHERS. “Benchmarking Large Language Models for Multi-Agent Systems: A Comparative Analysis of AutoGen, CrewAI, and TaskWeaver”. In: *Lecture Notes in Computer Science*, 2024. doi: 10.1007/978-3-031-70415-4_4.

[48] RAMIREZ-RUEDA, A., OTHERS. “Transforming Software Development: A Study on the Integration of Multi-Agent Systems and Large Language Models for Automation”. In: *2024 IEEE International Conference on Software Engineering (CONISOFT)*, 2024. doi: 10.1109/CONISOFT63288.2024.00013.

[49] LIN, Z., OTHERS. "Debate, Verify, and Debug: A Multi-Agent Planning Framework for Reliable Code Generation". In: *2025 IEEE International Conference on Computer and Communications Engineering Technology (CCET)*, 2025. doi: 10.1109/CCET66260.2025.11199679.

[50] KARABIYIK, O., OTHERS. "RefactorGPT: A ChatGPT-Based Multi-Agent Framework for Automated Code Refactoring", *PeerJ Computer Science*, 2025. doi: 10.7717/peerj-cs.3257.

[51] OTHERS. "DRF: LLM-Agent Dynamic Reputation Filtering Framework". In: *16th International Conference on Swarm Intelligence (ICSI 2025), Lecture Notes in Computer Science*, 2025. doi: 10.1007/978-981-95-4384-7_10.

[52] MANDULAPALLI, S., OTHERS. "Development of Agentic Workflows with LangGraph for Software Development Life Cycle Automation". In: *Lecture Notes in Computer Science*, 2025. doi: 10.1007/978-3-031-98235-4_4.

[53] PU, J., OTHERS. "Assistance or Disruption? Exploring and Evaluating the Design and Trade-offs of Proactive AI Programming Support". In: *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, 2025. doi: 10.1145/3706598.3713357.

[54] ZAMFIRESCU, C., OTHERS. "Beyond Code Generation: LLM-Supported Exploration of the Program Design Space". In: *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, 2025. doi: 10.1145/3706598.3714154.

[55] RAMIREZ, D., EKLER, P. "Exploring the Relationship Between Code Metrics and the Ability of Large Language Models to Solve Code Issues". In: *2025 IEEE 15th International Conference on Cognitive Infocommunications (CogInfoCom)*, 2025. doi: 10.1109/CogInfoCom66819.2025.11200869.

[56] ABIAKL, M., OTHERS. "NeSy Is Alive and Well: A LLM-Driven Symbolic Approach for Better Code Comment Data Generation and Classification". arXiv preprint, 2024. arXiv identifier to be confirmed by author.

[57] GROTOV, A., OTHERS. "Debug Smarter, Not Harder: AI Agents for Error Resolution in Computational Notebooks". In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP)*, 2024. doi: 10.18653/v1/2024.emnlp-demo.38.

[58] MÜNDLER, N., MÜLLER, M. N., HE, J., et al. "SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents". In: *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. Available at: <https://arxiv.org/abs/2406.12952>.

[59] LIU, M., OTHERS. "VVF-AI: A Vulnerability Verification Framework Based on AI-Agent". In: *2025 International Conference on Artificial Intelligence, Networking and Information Technology (AINIT)*, 2025. doi: 10.1109/AINIT65432.2025.11035850.

[60] YANG, H., OTHERS. "AutoVerus: Automated Proof Generation for Rust Code", *Proceedings of the ACM on Programming Languages*, 2025. doi: 10.1145/3763174.

[61] PALANIAPPAN, S., OTHERS. "Generative AI Tools for Accelerated Software Engineering", *Advances in Computers*, 2025. doi: 10.1016/bs.adcom.2025.06.008.

[62] REVURI, S., OTHERS. "Artificial Intelligence (AI) Technologies and Tools for Accelerated Software Development", *Advances in Computers*, 2025. doi: 10.1016/bs.adcom.2025.07.001.

[63] ZHAO, Y., OTHERS. "MAGE: A Multi-Agent Engine for Automated RTL Code Generation". In: *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, 2025. doi: 10.1109/DAC63849.2025.11133191.

[64] GUO, S., DENG, C., WEN, Y., et al. "DS-Agent: Automated Data Science by Empowering Large Language Models with Case-Based Reasoning". In: *Proceedings of the 41st International Conference on Machine Learning (ICML)*, Proceedings of Machine Learning Research, 2024.

[65] CARTAXO, B., PINTO, G., SOARES, S. "Rapid Reviews in Software Engineering". arXiv preprint arXiv:2003.10006, 2020. Available at: <https://arxiv.org/abs/2003.10006>.

[66] HEVNER, A. R., MARCH, S. T., PARK, J., et al. "Design Science in Information Systems Research", *MIS Quarterly*, v. 28, n. 1, pp. 75–105, 2004.

[67] MARCH, S. T., SMITH, G. F. "Design and Natural Science Research on Information Technology", *Decision Support Systems*, v. 15, n. 4, pp. 251–266, 1995.

[68] MCNEMAR, Q. "Note on the Sampling Error of the Difference between Correlated Proportions or Percentages", *Psychometrika*, v. 12, n. 2, pp. 153–157, 1947. doi: 10.1007/BF02295996.

[69] ROMANO, J., KROMREY, J. D., CORAGGIO, J., et al. "Appropriate Statistics for Ordinal Level Data". In: *Annual Meeting of the Florida Association of Institutional Research*, 2006.

[70] KRUSKAL, W. H., WALLIS, W. A. "Use of Ranks in One-Criterion Variance Analysis", *Journal of the American Statistical Association*, v. 47, n. 260, pp. 583–621, 1952. doi: 10.1080/01621459.1952.10483441.

[71] MANN, H. B., WHITNEY, D. R. "On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other", *The Annals of Mathematical Statistics*, v. 18, n. 1, pp. 50–60, 1947. doi: 10.1214/aoms/1177730491.

[72] LIU, J., SHEN, D., ZHANG, Y., et al. "What Makes Good In-Context Examples for GPT-3?" In: *Proceedings of the Workshop on Deep Learning for Low-Resource NLP (DeeLIO) at ACL '22*, pp. 100–114. Association for Computational Linguistics, 2022.

[73] ARCURI, A., BRIAND, L. "A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering", *Software Testing, Verification and Reliability*, v. 24, n. 3, pp. 219–250, 2014.

[74] HOLM, S. "A Simple Sequentially Rejective Multiple Test Procedure", *Scandinavian Journal of Statistics*, v. 6, n. 2, pp. 65–70, 1979.

[75] COHEN, J. *Statistical Power Analysis for the Behavioral Sciences*. 2nd ed. Hillsdale, NJ, USA, Lawrence Erlbaum Associates, 1988.

[76] HUANG, C., MA, A., YU, J., et al. "ACCESS: Prompt Engineering for Automated Web Accessibility Violation Corrections", *arXiv preprint arXiv:2401.16450*, 2024. Available at: <https://arxiv.org/abs/2401.16450>.

[77] TRICCO, A. C., LANGLOIS, E. V., STRAUS, S. E. (Eds.). *Rapid Reviews to Strengthen Health Policy and Systems: A Practical Guide*. Geneva, Switzerland, World Health Organization, 2017. Available at: <https://iris.who.int/items/d3786f07-fe20-402e-90e1-e5d7662dcb30>.

[78] COHEN, J. "A Coefficient of Agreement for Nominal Scales", *Educational and Psychological Measurement*, v. 20, n. 1, pp. 37–46, 1960. doi: 10.1177/001316446002000104.

Appendix A

Violation Taxonomy

This appendix provides the complete taxonomy of the seven WCAG violation categories used throughout the dissertation, including representative WCAG 2.1 success criteria, typical observable symptoms, canonical repair patterns, and the complexity rating that informs the adaptive routing decision.

Table A.1: Violation taxonomy (Part 1 of 2): categories `alt-text`, `label`, `aria`, and `keyboard`.

Type	WCAG Criteria	Observable Symptom	Canonical Repair Pattern	Complexity
`alt-text`	1.1.1 Non-text Content	`<img>` missing `alt`; or `alt=""` applied to an informative image that conveys meaning to the user	Add descriptive `alt` text; use `alt=""` for decorative images; apply `role="presentation"` when element must remain in DOM	**Simple**
`label`	1.3.1 Info & Relationships; 2.4.4 Link Purpose; 2.5.3 Label in Name	Generic link text (*click here*, *read more*); form control lacks an associated label; visible label absent from computed accessible name	Add `aria-label` or `aria-labelledby`; associate `<label htmlFor>`; append visually-hidden text via `.sr-only`	**Simple**–**Moderate**
`aria`	4.1.2 Name, Role, Value; 4.1.3 Status Messages; 1.3.5 Input Purpose	Invalid or redundant ARIA role; required state attribute absent (e.g., `aria-expanded` missing on disclosure widget); role conflicts with native element semantics	Set the correct `role`; add required ARIA state/property attributes; remove attributes that duplicate native HTML semantics	**Simple**
`keyboard`	2.1.1 Keyboard; 2.1.2 No Keyboard Trap; 2.4.1 Bypass Blocks	Non-interactive element receives or blocks focus; key event handlers missing on custom widgets; `tabIndex` misused or absent	Add `tabIndex="0"`; attach `onKeyDown`/`onKeyUp` implementing Space/Enter; replace non-interactive wrappers with `<button>` or `<a>`	**Moderate**

Table A.2: Violation taxonomy (Part 2 of 2): categories `focus`, `semantic`, and `contrast`.

Type	WCAG Criteria	Observable Symptom	Canonical Repair Pattern	Complexity
`focus`	2.4.7 Focus Visible (WCAG 2.1); 2.4.11 Focus Appearance (WCAG 2.2⋆)	CSS `outline: none` suppresses focus ring without replacement; focus indicator has insufficient contrast ratio; programmatic focus not moved to content after route change	Remove bare `outline: none`; add custom ring via `:focus-visible`; call `.focus()` on page landmark after navigation	**Moderate**
`semantic`	1.3.1 Info & Relationships; 1.3.2 Meaningful Sequence; 2.4.6 Headings & Labels	Heading levels skip (e.g., `h1` → `h3`); layout table used for non-tabular data; `<div>` or `<span>` used where a semantic element exists	Correct heading hierarchy; replace layout `<div>` with `<section>`, `<nav>`, `<main>`, or `<aside>`; restructure DOM to match visual reading order	**Moderate–Complex**
`contrast`	1.4.3 Contrast Minimum; 1.4.11 Non-text Contrast	Text-to-background ratio below 4.5:1 (normal text) or 3:1 (large text, $\geq$18 pt or 14 pt bold); UI component boundary below 3:1	Darken foreground or lighten background colour; update CSS custom properties or design tokens; verify propagation through theme system	**Complex**

* WCAG 2.4.11 (Focus Appearance) was introduced in WCAG 2.2 and falls outside the primary WCAG 2.1 AA scope of this dissertation. The `focus` category is defined in the taxonomy for completeness and for use in future evaluations targeting WCAG 2.2; no `focus` violations were present in the canonical benchmark corpus (experiment `8fdfb9f9`), which was scanned under WCAG 2.1 AA rules.

A.1 Mapping Tables

A.1.1 WCAG Criterion to Issue Type

Table A.3 provides the curated `WCAG_TO_ISSUE_TYPE` mapping used by the detection protocol to classify deduplicated issues. Criteria not listed are mapped via the rule-identifier table (Table A.4) or assigned the catch-all type `OTHER`.

Table A.3: Representative WCAG 2.1 success criteria to issue type mapping (40+ criteria; abbreviated here for readability).

WCAG SC	Name	Issue type
1.1.1	Non-text Content	`alt-text`
1.3.1	Info and Relationships	`semantic`
1.3.2	Meaningful Sequence	`semantic`
1.3.5	Identify Input Purpose	`aria`
1.4.3	Contrast (Minimum)	`contrast`
1.4.11	Non-text Contrast	`contrast`
2.1.1	Keyboard	`keyboard`
2.1.2	No Keyboard Trap	`keyboard`
2.4.1	Bypass Blocks	`keyboard`
2.4.4	Link Purpose (In Context)	`label`
2.4.6	Headings and Labels	`semantic`
2.4.7	Focus Visible	`focus`
2.4.11	Focus Appearance	`focus`
2.5.3	Label in Name	`label`
4.1.2	Name, Role, Value	`aria`
4.1.3	Status Messages	`aria`

A.1.2 Scanner Rule Identifier to Issue Type

Table A.4: Representative axe-core and Pa11y rule identifiers to issue type mapping (55+ rules; abbreviated).

Rule ID	**Tool**	**Issue type**
`image-alt`	axe-core	`alt-text`
`input-image-alt`	axe-core	`alt-text`
`color-contrast`	axe-core	`contrast`
`aria-required-attr`	axe-core	`aria`
`aria-valid-attr`	axe-core	`aria`
`aria-roles`	axe-core	`aria`
`label`	axe-core	`label`
`link-name`	axe-core	`label`
`button-name`	axe-core	`label`
`keyboard`	axe-core	`keyboard`
`focus-order-semantics`	axe-core	`focus`
`heading-order`	axe-core	`semantic`
`landmark-one-main`	axe-core	`semantic`
`WCAG2AA.1_1_1.H37`	Pa11y	`alt-text`
`WCAG2AA.1_4_3.G18`	Pa11y	`contrast`
`WCAG2AA.4_1_2.H91`	Pa11y	`aria`

Appendix B

Rapid Review Search Protocol

This appendix documents the complete search and screening protocol for the Rapid Review presented in Chapter 3, following the transparency requirements of the adopted RR methodology [65].

B.1 Search Query

The search was executed in **Scopus** on 15 February 2026. The query was constructed using a PICO-inspired structure adapted to software engineering reviews [43], combining three conceptual blocks: (i) AI/ML intervention, (ii) code repair/transformation outcomes, and (iii) agentic architecture as differentiator.

```
TITLE-ABS-KEY(
  (
    ("artificial intelligence" OR "AI" OR "machine learning"
     OR "deep learning" OR "neural network*")
    AND
    ("code repair" OR "code correction" OR "code modification"
     OR "code fixing" OR "bug fixing" OR "automated repair"
     OR "program repair" OR "code refactoring"
     OR "code generation" OR "code transformation")
    AND
    ("agent*" OR "autonomous agent*" OR "AI agent*"
     OR "intelligent agent*" OR "software agent*")
  )
)
AND PUBYEAR > 2019
AND (
  LIMIT-TO(SUBJAREA,"COMP")
```

```
  OR LIMIT-TO(SUBJAREA,"ENGI")
)
```

B.2 Eligibility Criteria

Inclusion Criteria

IC1 Studies proposing, evaluating, or analysing AI agents (including LLM-based agents) for code repair, bug fixing, refactoring, code transformation, or code generation with an explicit agentic control loop.

IC2 Publication year 2020–2026 (inclusive).

IC3 Indexed in Scopus under `COMP` or `ENGI` subject areas.

IC4 Peer-reviewed venue (conference, journal, book chapter), in English.

Exclusion Criteria

EC1 Proceedings-only records without an individually attributable paper.

EC2 Duplicates; only the most complete or most recent version retained.

EC3 Studies on agents in non-code domains (robotics, chemistry, navigation, non-code ML applications).

B.3 Screening Outcomes

Table B.1: Rapid Review screening summary (PRISMA-style).

Stage	Records	Cumulative excluded
Records retrieved from Scopus	138	–
After duplicate removal	120	18
After title/abstract screening	57	81
After full-text screening	55	83
Included in synthesis	**55**	–

Principal exclusion reasons at title/abstract stage: (i) “agent” used in non-AI meanings (networking, chemistry); (ii) AI applied to non-code tasks; (iii) HDL-only hardware work with limited software transferability; (iv) incomplete records without attributable paper.

B.4 Corpus Characteristics

Table B.2: Publication characteristics of the 55 included studies.

Characteristic	**Value**
Time window	2020–early 2026
Included studies	55
Conference papers	40 (72.7%)
Journal articles	12 (21.8%)
Books / book chapters	3 (5.5%)
Publications 2024–2025	49 (89.1%)
2021	1 (1.8%)
2023	1 (1.8%)
2024	23 (41.8%)
2025	26 (47.3%)
2026	4 (7.3%)

B.5 Included Primary Studies

The 55 studies that survived full-text screening are listed below, ordered by publication year (most recent first) and referenced as **[S1]**–**[S55]**. Each entry gives the lead authors, year, title, venue (where recorded by the indexing service), and DOI. The complete bibliographic records, including all authors, abstracts, and keywords as exported from Scopus, are bundled in the replication package as `Scopus.bib` (the raw database export) and `articles.xls` (the screening spreadsheet with per-record inclusion decisions), so that the selection can be reproduced or re-screened independently.

[S1] Coviello, Rao, Khojastepour et al. (2026). *Bifröst: Peer-to-Peer Load-Balancing for Function Execution in Agentic AI Systems.* Lecture Notes in Computer Science. `doi:10.1007/978-3-031-99854-6_19`

[S2] Lou, Hu, Ma et al. (2026). *DRF: LLM-AGENT Dynamic Reputation Filtering Framework.* Lecture Notes in Computer Science. `doi:10.1007/978-981-95-4384-7_10`

[S3] Mandulapalli, Hernandez, Hall et al. (2026). *Development of Agentic Workflows with LangGraph for Software Development Life Cycle*

Automation. Communications in Computer and Information Science. doi:10.1007/978-3-031-98235-4_4

[**S4**] Hajjarzadeh, Shakeri Hossein Abad, Far (2026). *LLM-Based MaSE, A Software Development Framework for Developing Multi-agent Systems*. Lecture Notes in Computer Science. doi:10.1007/978-981-96-8889-0_37

[**S5**] Rajendran, Besiahgari, Patil et al. (2025). *A Multi-Agent LLM Environment for Software Design and Refactoring: A Conceptual Framework*. Conference Proceedings - IEEE SOUTHEASTCON. doi:10.1109/SoutheastCon56624.2025.10971563

[**S6**] Nehzati (2025). *A quantum-inspired, biomimetic, and fractal framework for self-healing AI code generation: bridging responsible automation and emergent intelligence*. Frontiers in Artificial Intelligence. doi:10.3389/frai.2025.1662220

[**S7**] Baumann, Krämer (2025). *An LLM-Based Multi-Agent Framework for Evolutionary Blackbox Optimization*. doi:10.1145/3712255.3726575

[**S8**] Revuri, Sakthivel, Nagasubramanian (2025). *Artificial intelligence (AI) technologies and tools for accelerated software development*. Advances in Computers. doi:10.1016/bs.adcom.2025.07.001

[**S9**] Pu, Lazaro, Arawjo et al. (2025). *Assistance or Disruption? Exploring and Evaluating the Design and Trade-offs of Proactive AI Programming Support*. doi:10.1145/3706598.3713357

[**S10**] Yang, Li, Misu et al. (2025). *AutoVerus: Automated Proof Generation for Rust Code*. Proceedings of the ACM on Programming Languages. doi:10.1145/3763174

[**S11**] Barbarroxa, Gomes, Vale (2025). *Benchmarking Large Language Models for Multi-agent Systems: A Comparative Analysis of AutoGen, CrewAI, and TaskWeaver*. Lecture Notes in Computer Science. doi:10.1007/978-3-031-70415-4_4

[**S12**] Zamfirescu-Pereira, Jun, Terry et al. (2025). *Beyond Code Generation: LLM-supported Exploration of the Program Design Space*. doi:10.1145/3706598.3714154

[**S13**] Zhao, Jiang, Lee et al. (2025). *COMMIT0: LIBRARY GENERATION FROM SCRATCH*.

[**S14**] Zhang, Singhal, Zou et al. (2025). *Can AI Fix Buggy Code? Exploring the Use of Large Language Models in Automated Program Repair*. Computer.

doi:10.1109/MC.2025.3527407

[**S15**] Aldweesh, Alauthman, Alateef et al. (2025). *Charting the Uncharted: A Taxonomy of Emerging Cybersecurity Threats From Generative AI Systems.* doi:10.4018/979-8-3373-0832-6.ch002

[**S16**] Puvvadi, Kumar, Santoria et al. (2025). *Coding Agents: A Comprehensive Survey of Automated Bug Fixing Systems and Benchmarks.* doi:10.1109/CSNT64827.2025.10968728

[**S17**] Lin, Zou, Yuan (2025). *Debate, Verify, and Debug: A Multi-Agent Planning Framework for Reliable Code Generation.* International Conference on Computer and Communication Engineering Technology, CCET. doi:10.1109/CCET66260.2025.11199679

[**S18**] Esakkiraja, Akhiyarov, Shanmugham et al. (2025). *DeepCodeSeek: Real-Time API Retrieval for Context-Aware Code Generation.* CEUR Workshop Proceedings.

[**S19**] Song, Goknil, Jiang et al. (2025). *Developing Multi-Agent LLM Applications Through Continuous Human-LLM Co-Programming.* doi:10.1109/CAIN66642.2025.00013

[**S20**] Campbell, Chen, Luk et al. (2025). *Enhancing LLM-based Quantum Code Generation with Multi-Agent Optimization and Quantum Error Correction.* doi:10.1109/DAC63849.2025.11133316

[**S21**] Ramirez, Ekler (2025). *Exploring the Relationship Between Code Metrics and the Ability of Large Language Models to Solve Code Issues.* doi:10.1109/CogInfoCom66819.2025.11200869

[**S22**] Palaniappan, Prabha, Veerandeswari et al. (2025). *Generative AI tools for accelerated software engineering.* Advances in Computers. doi:10.1016/bs.adcom.2025.06.008

[**S23**] Ilager, Briem, Brandić (2025). *Green-Code: Learning to Optimize Energy Efficiency in Llm-Based Code Generation.* doi:10.1109/CCGRID64434.2025.00068

[**S24**] Wu, Fard (2025). *HumanEvalComm: Benchmarking the Communication Competence of Code Generation for LLMs and LLM Agents.* ACM Transactions on Software Engineering and Methodology. doi:10.1145/3715109

[**S25**] Zhao, Zhang, Huang et al. (2025). *MAGE: A Multi-Agent Engine for Automated RTL Code Generation.* doi:10.1109/DAC63849.2025.11133191

[**S26**] Akilesh, Sekar, Om Kumar et al. (2025). *Multi-Agent hierarchical workflow for autonomous code generation with Large Language Models.* doi:10.1109/SCEECS64059.2025.10940635

[**S27**] Sharanarthi, Polineni (2025). *Real-Time Adaptive Code Analysis with a Self-Learning Multi-Agent Framework: A Retrieval-Augmented Reinforcement Learning Approach.* doi:10.1109/ICAIDE65466.2025.11189485

[**S28**] Karabiyik (2025). *RefactorGPT: a ChatGPT-based multi-agent framework for automated code refactoring.* PeerJ Computer Science. doi:10.7717/peerj-cs.3257

[**S29**] Bouzenia, Devanbu, Pradel (2025). *RepairAgent: An Autonomous, LLM-Based Agent for Program Repair.* Proceedings - International Conference on Software Engineering. doi:10.1109/ICSE55347.2025.00157

[**S30**] Liu, Liu, Tang et al. (2025). *VVF-AI: A vulnerability verification framework based on AI- agent.* doi:10.1109/AINIT65432.2025.11035850

[**S31**] Noring, Jain, Fernandez et al. (2024). *AI-Assisted Programming for Web and Machine Learning: Improve your development workflow with ChatGPT and GitHub Copilot.*

[**S32**] Ramanan, Khan, Rao (2024). *ASPIRE: A Multi-Agent Framework for Execution-Free Code Analysis and Repair.* doi:10.1109/BigData62323.2024.10825553

[**S33**] Wong, Tan (2024). *Aligning Crowd-Sourced Human Feedback for Reinforcement Learning on Code Generation by Large Language Models.* IEEE Transactions on Big Data. doi:10.1109/TBDATA.2024.3524104

[**S34**] Zhang, Ruan, Fan et al. (2024). *AutoCodeRover: Autonomous Program Improvement.* doi:10.1145/3650212.3680384

[**S35**] Sukur, Milošević, Pracner et al. (2024). *Automated program improvement with reinforcement learning and graph neural networks.* Soft Computing. doi:10.1007/s00500-023-08559-1

[**S36**] Shetty, Chen, Somashekar et al. (2024). *Building AI Agents for Autonomous Clouds: Challenges and Design Principles.* doi:10.1145/3698038.3698525

[**S37**] Ramhorry (2024). *Building Intelligent Applications with Generative AI: Explore the potential of AI for next gen applications.*

[**S38**] Benaddi, Ouaddi, Souha et al. (2024). *Code Generation Templates for Accelerating Chatbot Development in Smart Tourism.* doi:10.1109/CommNet63022.2024.10793260

[**S39**] Venuto, Islam, Klissarov et al. (2024). *Code as Reward: Empowering Reinforcement Learning with VLMs*. Proceedings of Machine Learning Research.

[**S40**] Guo, Deng, Wen et al. (2024). *DS-Agent: Automated Data Science by Empowering Large Language Models with Case-Based Reasoning*. Proceedings of Machine Learning Research.

[**S41**] Grotov, Borzilov, Krivobok et al. (2024). *Debug Smarter, Not Harder: AI Agents for Error Resolution in Computational Notebooks*. doi:10.18653/v1/2024.emnlp-demo.38

[**S42**] Singh, Ehtesham, Kumar et al. (2024). *Enhancing AI Systems with Agentic Workflows Patterns in Large Language Model*. doi:10.1109/AIIoT61789.2024.10578990

[**S43**] Sakib, Khan, Karim (2024). *Extending the Frontier of ChatGPT: Code Generation and Debugging*. doi:10.1109/ICECET61485.2024.10698405

[**S44**] Chaplia, Klym (2024). *Extracting REST API Endpoints from Microservices Using LLM Agents*. doi:10.1109/DESSERT65323.2024.11122242

[**S45**] Lashuel, Kurdistan, Green et al. (2024). *LLM-Based Code Generation for Querying Temporal Tabular Financial Data*. doi:10.1109/CIFER62890.2024.10772804

[**S46**] Bajpai, Chopra, Biyani et al. (2024). *Let's Fix this Together: Conversational Debugging with GitHub Copilot*. Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC. doi:10.1109/VL/HCC60511.2024.00011

[**S47**] Abi-Akl (2024). *NeSy is alive and well: A LLM-driven symbolic approach for better code comment data generation and classification*. CEUR Workshop Proceedings.

[**S48**] Ding, Fan, Guehring et al. (2024). *Reasoning and Planning with Large Language Models in Code Development*. Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. doi:10.1145/3637528.3671452

[**S49**] Mündler, Müller, He et al. (2024). *SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents*. Advances in Neural Information Processing Systems.

[**S50**] Guo, Chen, Chen et al. (2024). *Smart Contract Code Repair Recommendation based on Reinforcement Learning and Multi-metric Opti-*

mization. ACM Transactions on Software Engineering and Methodology. doi:10.1145/3637229

[**S51**] Sales, Fontes Magalhães Mascarenhas, Simões et al. (2024). *Towards Automatic Code Generation for Robotic Soccer Behavior Simulation.* Journal of Intelligent and Robotic Systems: Theory and Applications. doi:10.1007/s10846-023-02036-5 *[Included under IC1: the study proposes an LLM-based agent with an explicit generation–execution–feedback loop for autonomous code synthesis. Although the application domain is robotic soccer simulation, the contribution is an agentic code-generation architecture and the domain is software code, not a robotic* control *agent, distinguishing it from the EC3 robotics-navigation exclusion. Retained for its architectural insights on agentic code generation.]*

[**S52**] Ramirez-Rueda, Benítez-Guerrero, Mezura-Godoy et al. (2024). *Transforming Software Development: A Study on the Integration of Multi-Agent Systems and Large Language Models for Automatic Code Generation.* doi:10.1109/CONISOFT63288.2024.00013

[**S53**] Stojanovic, Pavković, Četić et al. (2024). *Unit Test Generation Multi-Agent AI System for Enhancing Software Documentation and Code Coverage.* doi:10.1109/TELFOR63250.2024.10819096

[**S54**] Li, Ning (2023). *Autonomous GIS: the next-generation AI-powered GIS.* International Journal of Digital Earth. doi:10.1080/17538947.2023.2278895

[**S55**] Höhn, Faradouris (2021). *What Does It Cost to Deploy an XAI System: A Case Study in Legacy Systems.* Lecture Notes in Computer Science. doi:10.1007/978-3-030-82017-6_11

B.6 Sources Beyond the Systematic Search

The systematic search documented above is the evidence base for the Rapid Review synthesis in Chapter 3: no claim in that synthesis rests on a source outside the 55 studies in [S1]-[S55]. The dissertation as a whole, however, cites a number of works that the Scopus query did not return. This is expected and methodologically legitimate; to keep the boundary between *systematic evidence* and *supporting literature* explicit, those external references are obtained through three recognised, reproducible channels rather than ad-hoc collection.

Foundational and background literature. Seminal and survey works on automated program repair, web accessibility, and large language models, cited

mainly in Chapters 1 and 2 to establish concepts and situate the problem. These either predate or fall outside the *agentic-repair* scope of the search string (which deliberately required an explicit agent construct) and are used for conceptual grounding, not as primary evidence.

Citation chasing (unsystematic). A number of works cited for context were reached by following references in and out of the included studies. This was *not* conducted as a systematic snowballing pass in the sense of Kitchenham et al. [43] or of rapid-review guidance [65, 77]: there was no exhaustive sweep of the reference lists of the 55 included studies, no forward-citation query over the seed set, and consequently no enumerable set of studies "reached by snowballing" that could be reported as a distinct yield. Works obtained this way are used only to corroborate or contextualise findings that already rest on the systematic corpus; no conclusion of the synthesis depends on them. Stating this plainly is preferable to implying a procedural rigour the review did not exercise, and a systematic snowballing pass is listed among the review's extensions in Section 8.4.

Grey literature, standards, and tool documentation. Normative specifications (WCAG 2.1 [20], the Brazilian LBI [8], ABNT NBR 17225 [10]) and the official documentation or technical reports of the instruments the framework integrates (Pa11y [3], Axe-core [4], Playwright [22], ESLint-jsx-a11y [23], and the Qwen [13] and CodeLlama [12] model documentation). These are engineering artefacts cited as the authoritative description of an instrument or a legal requirement; by nature they are not indexed as peer-reviewed studies and so cannot surface in a database search, yet omitting them would leave the tooling and regulatory context undocumented.

This three-way separation follows the transparency expectations of the adopted rapid-review methodology [65]: the systematic corpus is reproducible from the artefacts in Section B.5, while every reference outside it is accounted for by one of the channels above.

B.7 Validity Threats of the Review

Search coverage. The query may miss works using alternative terminology ("LLM agent", "copilot", "code assistant") or works exhibiting agentic behaviour without the term "agent". Partially mitigated by broad AI terminology and by full-text screening for agentic properties.

Temporal boundary. The corpus reflects Scopus-indexed literature through early 2026. Given the dominance of 2024–2025 publications, the field is likely to evolve rapidly.

Synthesis level. Heterogeneity across tasks, benchmarks, and reporting conventions limits quantitative aggregation. The synthesis is thematic with descriptive statistics, consistent with RR practice [65].

Appendix C

Dataset Inclusion and Exclusion Criteria

This appendix documents the complete inclusion and exclusion criteria for repository selection, the snapshot protocol, and the annotation interface used to establish ground-truth labels.

C.1 Repository Selection Criteria

C.2 Stratified Sampling Design

Repositories are sampled across three stratification dimensions to ensure balanced coverage:

Application domain (7 cells) e-commerce, government, healthcare, education, developer tools, dashboard, social.

Component file count (3 cells) small (<50 files), medium (50–200), large (>200).

Repository popularity (3 cells) emerging (<500 stars), established (500–5,000), popular (>5,000).

The resulting $7 \times 3 \times 3 = 63$ strata are filled proportionally, targeting $N = 400$ repositories with a minimum of 2 per stratum to ensure coverage. Repositories are discovered via the GitHub Search API using the structured query:

```
language:TypeScript topic:react stars:>=50 pushed:>=[DATE-12mo]
NOT topic:template NOT topic:boilerplate NOT topic:starter
```

Discovery runs over a 10-day collection window to bound temporal variability.

Table C.1: Inclusion and exclusion criteria for the benchmark corpus.

ID	Type	Stage	Criterion
IC1	Include	Discovery	Repository contains ≥10 `.tsx` or `.jsx` component files
IC2	Include	Discovery	≥50 GitHub stars (community relevance signal)
IC3	Include	Discovery	At least one commit within the 12 months preceding data collection
IC4	Include	Discovery	Repository licensed under an OSI-approved open-source licence
IC5	Include	Scan	At least one accessibility issue detected by the scanner suite
EC1	Exclude	Discovery	Monorepo with no clearly bounded component sub-package
EC2	Exclude	Discovery	Component files are auto-generated (e.g., Storybook auto-docs, code-gen output)
EC3	Exclude	Discovery	Repository is a template, boilerplate, or scaffold project
EC4	Exclude	Scan	All scanner runs time out (>5 min per file)
EC5	Exclude	Discovery	Repository has a non-English primary README

C.3 Snapshot and Integrity Protocol

Each selected repository is snapshotted at the latest default-branch commit at the time of discovery. The protocol records:

1. The full SHA-1 commit hash (`pinned_commit`).
2. A complete archive of the repository file tree at that commit.
3. SHA-256 digests for all `.tsx`/`.jsx` component files.

All subsequent scans and experiments operate exclusively on the pinned snapshot, not on the live repository, ensuring exact reproducibility even if upstream repositories are modified or deleted. Snapshot integrity is validated by the QM6 metric ($\geq$90% of files with verified SHA-1).

C.4 Annotation Interface and Protocol

Ground-truth annotation is performed using the tool at `dataset/scripts/annotate.py`. For each finding, the interface displays:

- The scanner-produced finding object (violation code, message, selector).
- The surrounding code context (10 lines before and after the affected element, extracted from the pinned snapshot).
- The full text of the applicable WCAG 2.1 success criterion.
- A link to the W3C ACT rule associated with the violation, where applicable.

Annotators assign one of three labels:

`CONFIRMED` The violation is a genuine accessibility barrier in the source code as written.

`FALSE_POSITIVE` The scanner incorrectly flagged a compliant element; the code does not in fact violate the cited criterion.

`UNCERTAIN` Insufficient context to determine; this finding is excluded from the evaluation dataset.

Findings with $|T| \geq 2$ (detected by at least two independent scanners) and HIGH confidence are auto-accepted as `CONFIRMED` without human review. This threshold was validated against a random sample of 200 auto-accepted findings, achieving 98.5% precision on that calibration sample.

Note on the canonical experiment. In the canonical experiment (`8fdfb9f9`) reported in Chapter 6, the human annotation step was not executed. Instead, the auto-acceptance rule was extended to cover all MEDIUM-confidence violations (single-tool detections at critical or serious severity), in addition to HIGH-confidence violations. All 168 violations in the corpus were therefore accepted via severity-based auto-acceptance rather than human review. The planned annotation infrastructure (described above) is designed for use in future evaluation runs where LOW-confidence findings or disputed cases require adjudication. QM1 (inter-annotator agreement) is consequently not evaluable for the canonical experiment.

C.5 Dataset Quality Metrics

Table C.2: Dataset quality metrics (QM1–QM8) with thresholds and measurement procedures.

ID	Threshold	Description	Measurement
QM1	$\kappa \geq 0.70$	Inter-annotator agreement	Cohen's κ [78] on doubly-annotated sample
QM2	$N \geq 400$	Corpus size	Count of included repositories
QM3	No stratum >20%	Stratification balance	Maximum stratum fraction of total
QM4	All 7 types present	Violation type coverage	Distinct types in `CONFIRMED` set
QM5	FP rate ≤10%	False positive rate	Proportion of human-labelled `FALSE_POSITIVE`
QM6	≥90% with pinned commit	Snapshot integrity	Fraction of files with verified SHA-1 hash
QM7	≥70% scan success	Scan completeness	Fraction of files scanned without timeout/error
QM8	All 4 WCAG principles	Principle coverage	Distinct POUR principles in `CONFIRMED` set

Validation is automated in `dataset/scripts/validate.py` and can be executed with the `-strict` flag, which returns a non-zero exit code if any threshold is violated, enabling integration into a CI/CD quality gate. Meeting these thresholds is a necessary but not sufficient condition for dataset quality; they provide a reproducible minimum bar rather than a guarantee of representativeness.

Quality metrics for the canonical experiment. The canonical experiment (`8fdfb9f9`) does not satisfy three of the eight quality thresholds defined above, and this must be stated explicitly alongside the results in Chapter 6.

- **QM1** ($\kappa \geq 0.70$): not evaluable. The human annotation step was not executed (see note at end of Section C). All 168 violations were accepted via the severity-based auto-acceptance protocol. Inter-annotator agreement is therefore unknown for this corpus.
- **QM2** ($N \geq 400$): not met. The canonical corpus covers 36 repositories, approximately 9% of the target population of 400. The reduction is due to computational budget constraints detailed in Section 5.3.
- **QM5** (FP rate $\leq 10\%$): not evaluable. Without human annotation, the rate of false positives in the corpus cannot be quantified. The axe-core zero-false-positive guarantee at critical/serious severity provides partial mitigation for the subset of violations it detected, but does not extend to Pa11y or ESLint findings.

QM3, QM4, QM6, QM7, and QM8 are met by the canonical corpus. The failure to meet QM1, QM2, and QM5 is a scope limitation whose implications are discussed in Section 5.7.

Appendix D

Validation Pipeline Implementation

This appendix documents the complete four-layer validation pipeline, including the precise checks performed at each layer, the rejection and retry logic, and the configuration parameters available to practitioners.

D.1 Pipeline Overview

Every patch generated by a repair agent passes through four sequential validation layers before it is accepted and written to disk. Rejection at any layer triggers the retry mechanism. The overall pass criterion requires all four layers to return `PASS`.

These four layers establish that a patch is *safe to apply*; they do not, by themselves, establish that the violation was *resolved*. Resolution is confirmed separately: after an accepted patch is written, the framework re-runs the browser-based scanner suite and credits a violation as fixed only when the corresponding finding no longer appears, with credit bounded to the net per-criterion reduction so that a finding migrating to a new selector is not mistaken for a repair. That re-scan oracle, described in Section 4.5, populates the issue-fix-rate numerator; the pipeline documented below is the safety gate that precedes it.

D.2 Layer 1: Syntactic Validation

Checks

1. **Output schema validation.** The raw LLM output is parsed to locate exactly one fenced code block delimited by "‘`tsx` or "‘`typescript`. If zero or more than one block is found, the output is rejected.

Table D.1: Four-layer validation pipeline: layers, checks, and failure targets.

Layer	Name	Checks	Failure mode targeted
L1	Syntactic	Output schema; code block extraction; JSX well-formedness heuristic	Incomplete output; format violation; malformed JSX
L2	Structural Interface	Prop interface identity; export signatures; event handler presence	Functional regression; component rewrite
L3	Domain Compliance	Violation-type heuristics; required attribute presence	False resolution; ignored violation
L4	Code Quality	TypeScript compilation; complexity delta; added-line ratio	Over-engineering; type errors; code bloat

2. **Non-empty content.** The extracted block must be non-empty and must contain at least one `import` statement or JSX expression (heuristic: presence of `<` character followed by an identifier).

3. **Heuristic JSX well-formedness.** A lightweight check verifies: (a) balanced brace nesting (`{` and `}`); (b) no unclosed JSX tags detected via simple angle-bracket balance tracking; (c) no literal `"‘` sequences inside the code block (a common artefact of models that misformat their output).

This layer is intentionally lightweight: full TypeScript compilation is deferred to Layer 4 to minimise latency on the critical path.

Configuration

- `allowed_delimiters`: list of accepted fenced code block openers (default: `["‘tsx, "‘typescript, "‘jsx, "‘js]`).

D.3 Layer 2: Structural Interface Preservation

Rationale

Structural regressions are the highest-risk failure mode of automated repair: a patch that removes a prop, changes an export name, or omits an event handler will cause compile errors or silent runtime failures in consuming components. This layer verifies that the patch preserves the observable structural contract of the original component.

Scope limitation. This layer does not execute the repaired component against a test suite and does not verify runtime behaviour. Patches passing Layer 2 have preserved the component's structural interface; they have not been shown to be regression-free at runtime.

Checks

1. **Prop interface preservation.** The TypeScript interface definition of the component's props (extracted from the original file via lightweight AST analysis using `@typescript-eslint/parser`) must be structurally identical in the repaired file. Addition, removal, or type change of any prop property triggers rejection.

2. **Export signature preservation.** The component's default export and all named exports must be present in the repaired file with identical identifiers. Missing exports would cause import failures in consuming components.

3. **Event handler preservation.** All `onClick`, `onChange`, `onSubmit`, `onKeyDown`, `onKeyUp`, `onFocus`, `onBlur`, and other event handler identifier bindings present in the original file must be present in the repaired file. This check prevents the common failure mode where a model rewrites the component from scratch, omitting handlers it deems unnecessary.

Configuration

- `strict_prop_types`: if `true` (default), prop type changes trigger rejection in addition to additions/removals.

- `tracked_events`: list of event handler prefixes to track (default: all standard React event prefixes).

D.4 Layer 3: Domain Compliance

Rationale

A patch that passes syntactic and structural checks may still fail to address the actual violation. This layer applies heuristic pattern checks to verify that the patch includes the expected accessibility fix for each violation type.

Checks by Violation Type

`alt-text` The repaired file must contain an `alt` attribute on the element identified by the violation's CSS selector. The `alt` value must be non-empty (unless the image is confirmed decorative, in which case `alt=""` is valid).

`aria` The repaired file must contain the specific ARIA attribute flagged by the violation (e.g., `aria-label`, `role`, `aria-expanded`). The attribute value must be a non-empty string.

`label` The repaired file must contain an accessible name association: either `aria-label`, `aria-labelledby`, or a `<label>` element with a matching `htmlFor` attribute.

`contrast` The patch must modify at least one colour-related CSS property on the affected element: a colour hex value, a CSS variable assignment referencing a colour token, or a class name change where the new class contains a colour keyword.

`keyboard` The patch must add or modify at least one keyboard event handler (`onKeyDown`, `onKeyUp`) on the affected element, or must add a `tabIndex` attribute.

`focus` The patch must not introduce or leave unchanged a bare `outline: none` or `outline: 0` without a compensating focus style.

`semantic` The patch must change at least one element tag name or `role` attribute on the affected element. **Note:** this is the weakest of the domain checks. It accepts any tag substitution (including semantically equivalent ones, e.g., `<div>` → `<span>`) and does not verify that the chosen replacement actually satisfies the governing WCAG 1.3.1 criterion. Semantic violations are therefore the category most likely to pass Layer 3 while remaining unresolved by the browser re-scan oracle.

These checks are heuristic proxies, not full scanner re-evaluations. Full scanner re-evaluation is performed in the closed-loop metric computation described in Section 5.5.

D.5 Layer 4: Code Quality Assessment

Checks

1. **TypeScript compilation.** The repaired file is passed to the TypeScript compiler in `-noEmit` mode using the project's `tsconfig.json` (or a default

strict configuration if absent). Any type error triggers rejection. Type errors commonly indicate that the model introduced incorrect prop usage, invalid API calls, or undefined variable references.

2. **Complexity delta.** The cyclomatic complexity of the repaired file is computed using `complexity-report` and compared to the original. If the repaired file's complexity exceeds the original by more than $\Delta_{\max}$ (default: +3), the repair is rejected as over-engineered.

3. **Added code ratio.** The ratio of added lines to removed lines in the unified diff must not exceed $R_{\max}$ (default: 10×). Repairs adding ten times more code than they remove are unlikely to be minimal, violating the non-invasiveness principle.

Configuration Parameters

Table D.2: Configurable parameters for the validation pipeline.

Parameter	Default	Description
`max_complexity_delta`	3	Maximum allowed increase in cyclomatic complexity
`max_added_line_ratio`	10.0	Maximum ratio of added to removed lines
`strict_prop_types`	`true`	Reject on prop type changes
`allowed_delimiters`	`[tsx, typescript, jsx]`	Accepted code block openers
`ts_strict`	`false`	Enable TypeScript strict mode in compilation check

D.6 Retry and Fallback Logic

When a patch is rejected at any layer, the retry mechanism fires. Each retry attempt receives the validation failure as structured feedback appended to the original prompt:

```
// VALIDATION FAILURE --- Layer N: <LayerName>
// Reason: <specific failure description>
// Required: <what the patch must satisfy to pass>
// Previous attempt (rejected):
```tsx
<rejected patch>
```
```

‘ ‘ ‘

The retry loop is bounded by $R_{\max} = 3$ attempts (configurable). If all retries are exhausted for the primary agent, control passes to the fallback cascade:

1. **Primary agent** (selected by router): up to $R_{\max}$ attempts.
2. **DirectLLMAgent fallback**: up to $R_{\max}$ attempts with a simplified prompt (constraints-only, no few-shot examples). This fallback is invoked when the primary agent exhausts its retry budget without producing a patch that passes Layer 1 validation. See Section 4.4 for the routing context; this fallback cascade is part of the production pipeline but was not separately tracked in the experiment results.
3. **Skip**: the file is recorded as a repair failure with the last validation failure reason logged for analysis.

Appendix E

Framework Implementation Details

This appendix describes the software architecture, technology stack, and key implementation decisions of the *a11y-autofix* framework.

E.1 Technology Stack

Table E.1 lists the components the framework is built from, the specific technology chosen for each, and the role it plays in the pipeline. Versions are pinned in the replication package (Section E.6); the entries below record the major version under which every experiment reported in this dissertation was executed.

E.2 Module Structure

```
a11y-autofix/
|-- core/
|   |-- scanner.py          # MultiToolScanner: Pa11y + axe + Playwright
|   |-- detector.py         # DetectionProtocol: dedup, confidence, taxonomy
|   |-- prompt_builder.py   # PromptBuilder: 6-component template assembly
|   |-- router.py           # AdaptiveRouter: scoring matrix + agent selection
|   |-- agents/
|   |   |-- direct_llm.py   # DirectLLMAgent
|   |   |-- openhands.py    # OpenHandsAgent (HTTP REST client)
|   |   `-- swe_agent.py    # SWEAgent (CLI subprocess)
|   `-- validator.py        # ValidationPipeline: L1-L4 + retry logic
|-- experiment/
|   |-- runner.py           # ExperimentRunner: asyncio.gather + checkpointing
|   |-- metrics.py          # SR, IFR, MTTR, token cost, regression rate
```

Table E.1: Framework technology stack.

Component	**Technology**	**Role**
Runtime	Python 3.11	Orchestration, experiment runner, dataset tooling
LLM backend	Ollama	Local model serving (Q4_K_M quantisation)
Scanner: Pa11y	Node.js 20 + pa11y 9.1.1	HTML CodeSniffer-based static DOM scanning
Scanner: axe-core	axe-core 4.x (Node.js subprocess)	Zero-false-positive rule-based scanning
Scanner: dynamic	Playwright 1.x + axe-core	Rendered DOM scanning for React components
AST analysis	`@typescript-eslint/parser`	Prop interface and export extraction
TypeScript check	`tsc -noEmit`	Layer 4 type error detection
Complexity	`complexity-report`	Cyclomatic complexity delta
Async execution	`asyncio` + semaphore	Concurrent multi-model experiment runner
Embeddings	`sentence-transformers` (MiniLM-L6)	Few-shot example retrieval
Data format	JSON / JSONL	Scan results, experiment outputs, audit trail

```
|   `-- analysis.py         # Statistical tests, ablation, effect sizes
|-- dataset/
|   |-- collect.py          # GitHub API discovery + stratified sampling
|   |-- snapshot.py         # Pinned commit archiving + SHA-256 integrity
|   |-- scan.py             # Batch scanning for dataset construction
|   |-- annotate.py         # Two-annotator annotation interface (CLI)
|   `-- validate.py         # QM1-QM8 quality gate (--strict flag)
`-- config/
    |-- models.yaml         # Ollama model configurations
    |-- routing.yaml        # Router thresholds and scoring weights
    `-- validation.yaml     # Validation pipeline parameters
```

E.3 LLM Client Abstraction

All repair agents communicate with LLMs through a unified `LocalLLMClient` interface that wraps the Ollama HTTP API (`/api/generate`). Key implementation details:

- Temperature is fixed at $\theta = 0.1$ across all conditions.
- `num_predict` is set to 4,096 tokens (sufficient for a complete React component file).
- Responses are streamed to avoid HTTP timeout on large outputs; the client accumulates the stream and returns the complete text.
- A shared `asyncio.Semaphore` bounded by `max_concurrent_models` (default: 3) prevents GPU memory exhaustion when multiple Ollama instances are active simultaneously.

E.4 Experiment Checkpointing

The `ExperimentRunner` writes an intermediate JSON result file immediately after each model–strategy–file condition completes, encoding the condition parameters, repair outcome, validation layer results, token usage, and elapsed time. On restart, the runner reads all existing result files and skips any condition whose output file already exists, enabling seamless recovery from hardware failures, power outages, or memory errors without re-running completed work.

E.5 Few-Shot Example Library

The example library is a JSON file containing pre-curated violation–fix pairs organised by issue type. Each entry includes:

- `issue_type`: one of the seven taxonomy categories.
- `violation_description`: a structured description matching the format produced by Component 2 of the prompt template.
- `original_code`: the non-compliant JSX snippet.
- `repaired_code`: the compliant JSX snippet.
- `annotation`: a brief explanation of the change and the WCAG criterion it satisfies.
- `embedding`: a precomputed 384-dimensional vector from MiniLM-L6-v2 over the violation description text, used for similarity retrieval.

At runtime, the `PromptBuilder` selects the top-k examples (default: $k = 2$) by cosine similarity between the current violation's embedding and the library entries, restricted to the matching issue type.

E.6 Replication Package

The complete replication package is available at `https://github.com/jpwieland/a11y_experiment` and includes all artefacts required to reproduce the canonical experiment and extend it to new models or violation corpora. The repository was made publicly accessible prior to the submission of this dissertation; readers encountering access issues should contact the corresponding author, as the repository may have been transferred or renamed after submission.

Repository Structure

`framework/` Full Python source for the *a11y-autofix* framework: detection orchestration, prompt builder, SWE-agent integration, OpenHands integration, four-layer validation pipeline, and experiment runner. Entry point: `framework/run_experiment.py`.

`prompts/` All six-component prompt templates in versioned JSON format. Each template file specifies the role definition, violation context schema, code context schema, constraint list, output format specification, and exemplar selection configuration.

few_shot_library/ The few-shot exemplar library used in experiment 8fdfb9f9: 42 human-verified violation–fix pairs across all seven violation categories, with precomputed all-MiniLM-L6-v2 sentence embeddings (384-dimensional) stored as a NumPy .npy array and an accompanying JSON index.

dataset/ Benchmark corpus: per-project subdirectories with pinned commit snapshots (snapshots/), scan results per project (scan_results/), and a manifest (corpus_manifest.json) listing all 36 projects with their GitHub URL, pinned SHA, domain tier, size tier, and popularity tier. Repository contents are included under each project's original open-source licence.

experiment-results/ Raw output of experiment 8fdfb9f9: per-repetition experiment_result.json files (three repetitions × three models), aggregated repetitions_summary.json, and per-file metrics CSV.

analysis/ Jupyter notebooks that reproduce every table and figure in Chapter 6 from the raw experiment JSON: primary_metrics.ipynb (Tables 5.3–5.5, Figures 5.1–5.4), category_analysis.ipynb (Table 5.9), confidence_intervals.ipynb (Tables 5.10–5.11), and failure_taxonomy.ipynb (Table 6.1).

validation/ Standalone implementation of the four-layer validation pipeline with unit tests and a CLI harness for evaluating individual patches outside the experiment runner.

README.md Step-by-step reproduction guide covering: (1) hardware and software prerequisites (NVIDIA GPU with ≥6 GB VRAM, Python 3.10+, Ollama, Node.js 18+); (2) model download commands for each of the three evaluated models; (3) corpus reconstruction from manifest; (4) single-model experiment invocation; (5) analysis notebook execution order.

Hardware Requirements

The canonical experiment was executed on an Acer Nitro ANV15-52 laptop running Ubuntu 24.04 LTS: an NVIDIA GeForce RTX 4050 Mobile GPU (6 GB GDDR6 VRAM), a 13th-generation Intel Core i7-13620H CPU (10 cores / 16 threads), 32 GB DDR5 RAM, and a 1 TB NVMe SSD. The framework can run on any CUDA-capable GPU with ≥6 GB VRAM (the ≤7B Q4_K_M models load in roughly 2–4 GB). CPU-only inference is supported via Ollama's CPU backend but is not recommended for the 7B models (estimated ∼10× latency increase). A complete single-model three-repetition run on the 1,249-file corpus takes approximately 20 minutes for qwen2.5-coder-3b, 1 hour for qwen2.5-coder-7b, and 6 hours for codellama-7b

on the reference hardware, consistent with the per-file mean times reported in Chapter 6.

Extending to New Models or Corpora

To evaluate a new model: install it via `ollama pull <model>`, add an entry to `config/models.json` with the model identifier and context window size, and rerun `run_experiment.py`. No other configuration is required.

To evaluate on a new violation corpus: populate `dataset/snapshots/` with the target project, run `framework/scan.py` to generate a `scan_result.json`, add an entry to `corpus_manifest.json`, and run the experiment as normal. The framework adapts to any React/TypeScript project detectable by its four-tool scanning stack.

To adapt to a non-accessibility quality domain: replace the violation context schema (`prompts/violation_context.json`), the domain compliance layer (`validation/domain_compliance.py`), and the few-shot library with domain-appropriate examples. The remaining components (prompt builder, agent routing, structural validation, experiment runner, and analysis notebooks) require no modification.