SCALABLE SOFTWARE TESTING AND ANALYSIS WITH NLP METHODS

by

Guanqun Yang

A DISSERTATION

Submitted to the Faculty of the Stevens Institute of Technology
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Guanqun Yang, Candidate

ADVISORY COMMITTEE

Xueqing Liu, Chair Date

Hui Wang Date

William Eiers Date

Wei Yang Date

STEVENS INSTITUTE OF TECHNOLOGY
Castle Point on Hudson
Hoboken, NJ 07030
2026

©2026, Guanqun Yang. All rights reserved.

SCALABLE SOFTWARE TESTING AND ANALYSIS WITH NLP METHODS

ABSTRACT

Building reliable and secure software depends on work that outgrows the people available to do it: sorting security vulnerability reports, testing machine learning models for hidden failures, and checking the AI assistants that now help write code. All are still done largely by hand. This dissertation shows that natural language processing (NLP) methods can take over much of this work, through four studies arranged on two axes: where each study acts in the software development life cycle, from managing vulnerabilities in released software, to testing model behavior, to checking code as it is written; and how much independence the language model is given, rising through four stages, EXTRACT, GENERATE, ACT, and COLLABORATE.

FewVuln (EXTRACT) reads vulnerability reports and pulls out the affected software and versions, matching earlier systems with about 10% as many labeled examples. TestAug and HateModerate (GENERATE) write behavioral tests for text classifiers, exposing failures that a single accuracy score hides: TestAug cuts manual test authoring by over 98%, and HateModerate aligns a hate speech classifier with a platform's 41 written content policies without lowering its accuracy on ordinary inputs. PatchHolmes (ACT) searches a repository for the exact commit that fixes a vulnerability, a link most reports never record, raising the rate at which the correct fix is ranked first (Recall@1) by 25.34% over a strong method. WatchPoint (COLLABORATE) acts as a simulated developer, running diagnostic scripts against a live application and reporting what it observes so a coding assistant can repair its own work, recovering 57.6% of failing tasks, close to the 54.5% human rate.

Across the four studies, what makes these methods practical is how the model is

applied, through adaptation, prompting, tool use, and feedback. PatchHolmes tested this most directly: replacing its model with one about seven times smaller changes the outcome by under 1%, while changing how the model is used changes it by 25.34%. As AI writes more of the software it is meant to check, testing and analysis must move into the development loop and scale with the software rather than with the experts free to check it.

Author: Guanqun Yang

Advisor: Xueqing Liu

Date: July 7, 2026

Department: Computer Science

Degree: Doctor of Philosophy

Table of Contents

Abstract iii

List of Tables xi

List of Figures xvii

1 Introduction 1

1.1 Contributions 5

1.2 Dissertation Organization 7

2 Background 9

2.1 Software Testing 9

2.1.1 Traditional Software Testing 10

2.1.2 Testing Machine Learning Systems 12

2.1.3 Capability-Based Testing for Machine Learning Models 13

2.1.4 Testing AI Coding Assistants in the Loop 14

2.2 Software Analysis 17

2.2.1 Software Analysis Tasks 17

2.2.2 Security Vulnerability Reports 18

2.2.3 Information Extraction from Vulnerability Reports 19

2.2.4 Linking Vulnerabilities to Fixes 21

2.3 NLP Methods for Scalable Software Testing and Analysis 22

2.3.1 Fine-Tuning Pre-trained Language Models 22

2.3.2 In-Context Learning 24

2.3.3 Tool-Using Language Model Agents 25

2.3.4 Language-Model-Based Simulation and Evaluation 26
2.3.5 Why NLP Methods Make Testing and Analysis Scalable 27
2.4 Summary 29

3 Extracting Structured Facts from Security Vulnerability Reports 31
3.1 Introduction 31
3.2 Problem Definition and Challenges 35
3.2.1 Few-Sample Named Entity Recognition 35
3.2.2 Named Entity Recognition for Vulnerability Reports 35
3.2.3 Data-Specific Challenges 36
3.3 Methods 37
3.3.1 Fine-Tuning Pre-trained Language Models 37
3.3.2 Few-shot Named Entity Recognition 39
3.4 Experiments 40
3.4.1 Datasets 40
3.4.2 Evaluation Metrics 41
3.4.3 Experimental Setup 41
3.4.4 Experimental Results: Fine-Tuning on `memc` 42
3.4.5 Experimental Results: Transfer Learning on Other Categories 44
3.5 Related Work 48
3.5.1 Information Extraction in Public Vulnerability Databases 48
3.5.2 Named Entity Recognition for Computer Security 48
3.5.3 Few-Sample Named Entity Recognition 49
3.6 Conclusions and Future Work 50

4 Generating Behavioral Tests for NLP Models 52
4.1 Introduction 52

4.2 Problem Definition 54
4.3 The TestAug Framework 55
4.3.1 Prompt Engineering for GPT-3 55
4.3.2 Filtering Incorrect Test Cases 56
4.3.3 Expanding GPT-3 Cases into Templates 57
4.4 Experiments 57
4.4.1 Experimental Settings 57
4.4.2 Evaluating Bug Detection Ability 58
4.4.3 Evaluating Diversity 58
4.4.4 Evaluating Validity and Manual Effort Saved 60
4.5 Application: Testing Hate Speech Detectors Against Content Policies 61
4.5.1 The HateModerate Framework 62
4.5.2 Testing Hate Speech Detectors 63
4.5.3 Improving Conformity Through Fine-Tuning 65
4.6 Related Work 68
4.6.1 Capability-Based Testing for NLP Models 68
4.6.2 Content Moderation and Hate Speech Detection 68
4.7 Conclusions and Future Work 69

5 Acting on Code Repositories to Retrieve Vulnerability Patches 70
5.1 Introduction 70
5.2 The PatchHolmes Method 74
5.2.1 Phase 1: Hybrid Retrieval 75
5.2.2 Phase 2: Agentic Listwise Selection 76
5.3 Experiments 78
5.3.1 Experimental Setup 78

5.3.2 Main Results 79
5.3.3 Ablation 81
5.3.4 Analysis: The Scaffolding, Not the Model, Drives the Result 82
5.4 Related Work 83
5.5 Discussion and Limitations 84
5.6 Conclusion 85

6 Collaborating with Coding Agents through Executable Feedback 86
6.1 Introduction 86
6.2 The WatchPoint System 88
6.2.1 The Coding Environment and Metrics 89
6.2.2 The Coding Workflow 90
6.2.3 The Simulated User 90
6.2.4 Contrast with Prior Feedback 92
6.3 Experiments 93
6.3.1 Experimental Setup 93
6.3.2 Performance on Web-Bench 93
6.3.3 Effectiveness of the Simulated User 94
6.3.4 Robustness of the Gains 96
6.3.5 Comparison with Human Feedback 98
6.4 Related Work 99
6.5 Discussion and Limitations 100
6.6 Conclusion 102

7 Conclusion 103
7.1 Summary of Contributions 103
7.2 Broader Implications 105

A Appendix **1**

A.1 FewVuln: Supplementary Materials 1

A.1.1 Dataset Statistics 1

A.2 TestAug: Supplementary Materials 4

A.2.1 Prompt Designs 4

A.2.2 Fine-Tuning Hyperparameters 5

A.2.3 Data Annotation Interface 5

A.2.4 Linguistic Capabilities 5

A.2.5 Evaluated Models 10

A.2.6 Validity and Classifier Performance 10

A.2.7 Template Expansion 15

A.3 HateModerate Application: Supplementary Materials 15

A.3.1 The Classes of Non-Hateful Examples 15

A.3.2 The List of Evaluated Hate Speech Models 15

A.3.3 The List of Training Datasets for CardiffNLP's Model 17

A.3.4 Excluding Sentences to Prevent Data Contamination 18

A.3.5 Fine-Tuning Hyperparameters and Process 18

A.3.6 Significance Tests for Fine-Tuning Experiments 19

A.3.7 Comparative Analysis of HateModerate and DynaHate 19

A.3.8 Overview of Facebook's Hate Speech Community Standards 22

A.4 PatchHolmes: Supplementary Materials 27

A.4.1 Sampling Validation for GitHubAD 27

A.4.2 Phase 1 Dense Embedder Pilot 29

A.4.3 Diff-Render Layer 30

A.4.4 Phase 2 System Prompt 31

A.4.5 IRCoT Scale-Stability Check 31

A.4.6 Parameter Regime of the Backbone Swap 33
A.4.7 Baseline Configurations 33
A.4.8 Metric Conventions 34
A.4.9 Hyperparameters 34
A.5 WatchPoint: Supplementary Materials 34
A.5.1 Retry and Feedback Mechanism 35
A.5.2 Prompts 36
A.5.3 Evaluation Infrastructure 38
A.5.4 Project Classification 38
A.5.5 Model Hyperparameters 39
A.5.6 Diagnostic Overhead 39
A.5.7 Single-Shot Design: Why Not Interpret the Observations 40
A.5.8 Run-to-Run Variance 41
A.5.9 Per-Project Results 42
A.5.10 User Study Protocol 43
A.5.11 Diagnostic Model Comparison 43
A.6 Model Identifiers Across Chapters 45

Bibliography **46**

Vita **59**

List of Tables

3.1 The proportion of sentences in the training set that include only non-entities 37

3.2 Statistics of the VIEM dataset 40

3.3 Fine-tuning results for the memc category 42

3.4 Transfer learning results averaged across 12 categories 45

4.1 Comparison of bug detection ability between TestAug and CheckList. Each cell shows the failure rate (error rate on the held-out validation set). 59

4.2 Linguistic diversity of test suites 59

4.3 TestAug saves manual efforts in generating new test cases 61

4.4 Failure rates of fine-tuning with and without HateModerate 66

4.5 Simple test cases for measuring bias. Each number is the probability of hate assigned by the fine-tuned model to the example. 67

5.1 Head-to-head ranking metrics on GitHubAD with Qwen3-235B. All numbers are percentages over the 809-vulnerability working subset; a method that errors on a vulnerability counts it as a miss. PatchHolmes's inspection list is extended to length 10 for a fair comparison against Favia's ten-deep output. † IRCoT returns a single pick, so its scores are equal across K. 79

5.2 Cross-corpus evaluation on PatchFinder_top10 under the settings of Storhaug et al. (2026). All numbers are percentages relative to the 1,252-vulnerability benchmark, using the same Qwen3-235B backbone. The PatchFinder row reports PatchFinder's own ten-rank ordering, so its Recall@1 (24.28%) is its literal top-1 pick. The hard Recall@10 ceiling is 46.09%: only 577 of the 1,252 vulnerabilities have the correct fix anywhere in the supplied pool, so the other 675 cannot be recovered by any single-pick method. 80

5.3 Ablation on GitHubAD with Qwen3-235B, in percentages over the same 809-vulnerability subset as Table 5.1. Rows A, B, and C replace the Phase 1 fusion with a single retriever path; row D replaces the short vulnerability description with the full database report (mean 1,786 against about 150 characters). 81

5.4 Backbone-scale check on GitHubAD across two backbones. 82

6.1 Aggregate results on Web-Bench. Pass@1 is the first-attempt pass rate; Pass@2 is the pass rate after at most one retry. The best agentic Pass@1 and Pass@2 are in bold. 94

6.2 Per-category results: the MiniMax M2.7 baseline (C1) against MiniMax M2.7 with a GLM-5 simulated user (C2). "ON/OFF" marks whether the simulated user is enabled. Δ is the change in Pass@2 in percentage points. "Retried" counts tasks that failed the first attempt and were retried (not skipped); "Rec." counts those that then passed; "Rate" is their ratio. 95

6.3 Run-to-run variance on the enabled categories at temperature zero. The difference between C2 and C1 at the multi-seed mean is 4.2 percentage points, against the 2.7 reported from a single seed in Table 6.2. The standard-deviation column reports the typical spread of a single seed's result around the mean. 96

6.4 Four diagnostic tiers on a common MiniMax M2.7 coding model. "Diff. vs. C1" is the change in on-category Pass@2 in percentage points. The useful range covers the mid tiers (GLM-5 and GPT-5.4); a top-tier diagnostic (Sonnet-4.6) and a bottom-tier one (Haiku-4.5) both fall within the seed-to-seed variation. 97

6.5 User study: two human testers against WatchPoint, in tasks passed within two attempts (out of 20 per project). Two graduate students replace the simulated user on pure HTML, CSS, and JavaScript projects that render in a browser without a build step. 98

A.1 Detailed statistics of VIEM dataset. 1

A.2 Prompt designs for paraphrase detection and natural language inference tasks. 4

A.3 Hyperparameter choice for model fine-tuning. 5

A.4 Linguistic capabilities that appeared in the experiments and their explanations with corresponding examples. The description of the tested linguistic capabilities follow the taxonomy and definition provided in previous work (Ribeiro et al., 2020; Tarunesh et al., 2021; Joshi et al., 2020). 6

A.5 The fine-tuned models we evaluated in this paper. 10

A.6 Proportion of valid sentences and performance of trained classifiers for automatic filtering. 10

A.7 Creating new templates based on test cases generated by GPT-3. 15

A.9 Failure rates of ① and ② on all categories of HateModerate. ①: Fine-tuning with CardiffNLP data + DynaHate. ②: Fine-tuning with CardiffNLP data + DynaHate + (HateModerate - DynaHate). 19

A.8 The statistics and p-values of the paired t-tests for comparing fine tuning with and without HateModerate. 23

A.10 Full name and description for Facebook's Hate Speech Community Standards (Facebook, 2022). We show matching names of guidelines and their index in Figure 4.5, the full descriptions of them are following. 23

A.11 Goodness-of-fit of the sampled GitHubAD working subset. 28

A.12 Phase 1 retrieval-only Recall@K with different dense embedders, in percentages over the 809-vulnerability subset. Only the dense embedder changes; the lexical path, the fusion, and the Phase 2 agent are unchanged. 30

A.13 Worked example for `CVE-2015-9251` (jQuery), correct commit `b078a62`: the three touched files and their priorities under Algorithm 2. The fix lives in `src/ajax/script.js`, which scores highest because it pairs the source tag with the largest non-test edit; `test/unit/ajax.js` is deprioritized despite its larger edit because the test tag carries a smaller base. 30

A.14 Scale stability of the IRCoT baseline: Recall@1 (in percentages) on the 809-vulnerability GitHubAD subset and on the full 8,401-vulnerability source corpus, across both backbones. Because IRCoT returns a single pick, Recall@1 equals Hit@1 here. The largest swing is 1.36%, so the baseline is stable across corpus scale. 31

A.15 Parameter regime of the backbone swap. Top block: the two backbones compared in Section 5.3.4. Bottom block: comparator counts from the Qwen3-Coder-Next technical report. Both sit in a near-tenfold parameter ratio, ours at 7.3 times active parameters; the score gap is small in both. 33

A.16 Hyperparameters used in PatchHolmes. All values are codebase defaults, with no per-vulnerability tuning. 35

A.17 Model hyperparameters for all six configurations. "n/a" marks a parameter that does not apply. All coding models use temperature zero with extended reasoning disabled. 39

A.18 Diagnostic-script execution on the enabled-category retried tasks (C2). Each cell is the fraction of retried tasks for which that agent produced at least one valid observation. 40

A.19 Largest first-attempt pass-rate differences between C1 and C2, which share the same coding model at temperature zero. Because the simulated user cannot affect the first attempt, every difference here is run-to-run variance. 41

A.20 Per-project Pass@2 (tasks out of 20) for the MiniMax M2.7 baseline (C1) and MiniMax M2.7 with a GLM-5 simulated user (C2), sorted by the change within each category. 42

A.21 All human feedback texts from the user study. "Rec.?" marks whether the task was recovered (the first attempt failed and the second passed after the feedback). Participants are anonymized as A and B. 44

A.22 Canonical provider identifiers for the models used in this dissertation. The main text refers to each model by its plain name; this table is the single place the exact identifiers appear. For the WatchPoint models (Chapter 6), the exact API checkpoint strings were not recorded in the source, so the table lists each model's role instead. 45

List of Figures

1.1 The four studies in this dissertation, placed on two axes. The vertical axis is autonomy: how much of the task the language model carries on its own, rising through EXTRACT (reporting facts), GENERATE (writing tests), ACT (using tools), and COLLABORATE (working with another agent). The horizontal axis is position in the software development life cycle: FewVuln and PatchHolmes manage vulnerabilities in existing software (security analysis and maintenance), TestAug and HateModerate generate behavioral tests (testing), and WatchPoint supplies executable feedback inside the coding loop, where AI agents now write software. Together the four studies span the quality-relevant back half of the life cycle. Each study's headline result appears beside it and is grounded in the corresponding chapter. 2

2.1 Software testing along a single continuum, unboxing the testing and implementation stages of the software development life cycle from Figure 1.1. Testing runs from traditional testing of a finished program, to capability-based testing of a finished machine learning model (TestAug and HateModerate), to executable testing inside the coding loop where an AI agent writes the software (WatchPoint), moving from a final gate on a finished artifact to an interactive signal inside the development loop. 10

2.2 Software analysis as security vulnerability management, unboxing the maintenance stage of the software development life cycle from Figure 1.1. A vulnerability described in free-form prose is read so that the affected software and versions are extracted (FewVuln), and then the commit that repairs it is located among the millions in the repository (PatchHolmes). 16

2.3 How NLP methods make testing and analysis scalable. Each method amplifies a small, fixed human input into coverage that grows with the software: a few labeled examples, a few seed tests, a goal with a few tools, and one simulated tester become, respectively, thousands of vulnerability reports read (FewVuln), hundreds of behavioral tests (TestAug and HateModerate), millions of commits searched (PatchHolmes), and repeatable in-the-loop testing (WatchPoint), across the quality-critical back half of the software development life cycle (Figure 1.1). 28

3.1 An example of a public security vulnerability report (CVE-2015-2384) 35

3.2 The t-SNE visualization of the `memc` training set at the token level before (left) and after (right) fine-tuning. Fine-tuning produces more compact clusters for both `SN` and `SV`. 43

3.3 The influence of training set size on fine-tuning performance 44

3.4 The influence of training set size on transfer learning performance 46

3.5 `StructShot` is sensitive to adversarial samples. When testing on the `csrf` category, an additional sample in the support set leads to 70% F1 score degradation for `SN`. 47

4.1 The control-flow graph of TestAug 55

4.2 Prompt design for sentiment classification test case generation. The test description specifies context; seed sentences help GPT-3 generate similar yet diverse cases. 56

4.3 Capability-wise error rates on paraphrase detection and natural language inference tasks. 60

4.4 The workflow of HateModerate data collection 62

4.5 Failure rates for hateful (right-facing bars) and non-hateful (left-facing bars) examples across each of Facebook's 41 policies 64

4.6 The comparison of failure rates across the sub-categories of non-hateful examples. 66

5.1 Listwise selection (PatchHolmes) recovers a pointwise failure (Favia) on `CVE-2015-9251`, a jQuery cross-site scripting flaw through cross-domain Ajax. On the same ten-candidate shortlist (the correct fix sits at rank 9), Favia assigns the same confidence to all ten candidates and picks the rank-1 commit, an unrelated refactor. PatchHolmes's agent instead reads five commits in shortlist order (1, 2, **9**, 4, 3) and identifies the one that adds the line blocking automatic execution of `text/javascript` responses. 71

5.2 PatchHolmes traces a vulnerability to its fix commit in two phases. Panels (a, b) plot accuracy against the cost in language-model calls; panel (c) is the system diagram. On GitHubAD (Liu et al., 2025), PatchHolmes (gold star) reaches Recall@1 of 59.95% with one agent conversation per vulnerability, against Favia's 34.61% at ten conversations. On the PatchFinder_top10 benchmark (Storhaug et al., 2026), PatchHolmes reaches 39.86% against PatchFinder's 24.28%, approaching the 46.09% recoverable ceiling. Phase 1 fuses a lexical retriever and a dense retriever into a top-100 candidate set; Phase 2 is an agent that drives four tools over up to 15 turns and submits one best commit, usually after reading 3 to 10 candidates. 74

6.1 Overview of the WatchPoint workflow. (a) The coding agent receives a task, edits code over several turns, and runs the end-to-end tests. On failure, WatchPoint generates diagnostic feedback through two parallel agents. (b) The diagnostic mechanism: a browser agent runs a script that inspects the live page, a terminal agent runs a script that checks the workspace and server state, and the combined structured observations are returned to the coding model for a second attempt. 89

A.1 The command line interface for data annotation. Annotators are given a test and three associated test cases from the template-based test suite; they are asked to annotate the validity of the GPT-3-generated sentences. Annotators are reminded of the guidelines for filtering invalid samples when labeling each sentence (shown at the top of the interface). We communicated explicitly for the intended uses of the annotated datasets before the annotation. 5

Chapter 1

Introduction

Software has become the infrastructure of everyday life, and the cost of getting it wrong is now measured in trillions of dollars. The Consortium for Information and Software Quality estimates that poor software quality cost the United States about $2.41 trillion in 2022, counting operational failures, unsuccessful projects, and the accumulated debt of defects left unfixed (Krasner, 2022). Two forces drive this figure, and both continue to grow. The first is insecure software. Most successful cyberattacks exploit a vulnerability that was already disclosed and already had a fix available, yet more than half of organizations cannot apply a fix for a critical vulnerability within the recommended 72 hours (Dissanayake et al., 2021), and the volume of disclosures keeps breaking records: the National Vulnerability Database recorded 40,009 new entries in 2024, an increase of 38.8% over the 28,818 of 2023 and the seventh consecutive record year (MITRE Corporation, 2024; National Institute of Standards and Technology, 2024). The second force is the spread of machine learning inside ordinary software. A learned model has no specification to test against; its mistakes are spread out and hidden behind a single accuracy number, and its bugs often live in the data rather than the code, so the methods that catch traditional defects do not catch these (Zhang et al., 2022).

Two Ways to Check Software, and a New Third. Software analysis and software testing are the two long-standing ways of checking software for vulnerabilities and defects, joined recently by a third. All three are among the most expensive parts of software development (Omri and Sinz, 2021), and all are still largely carried out

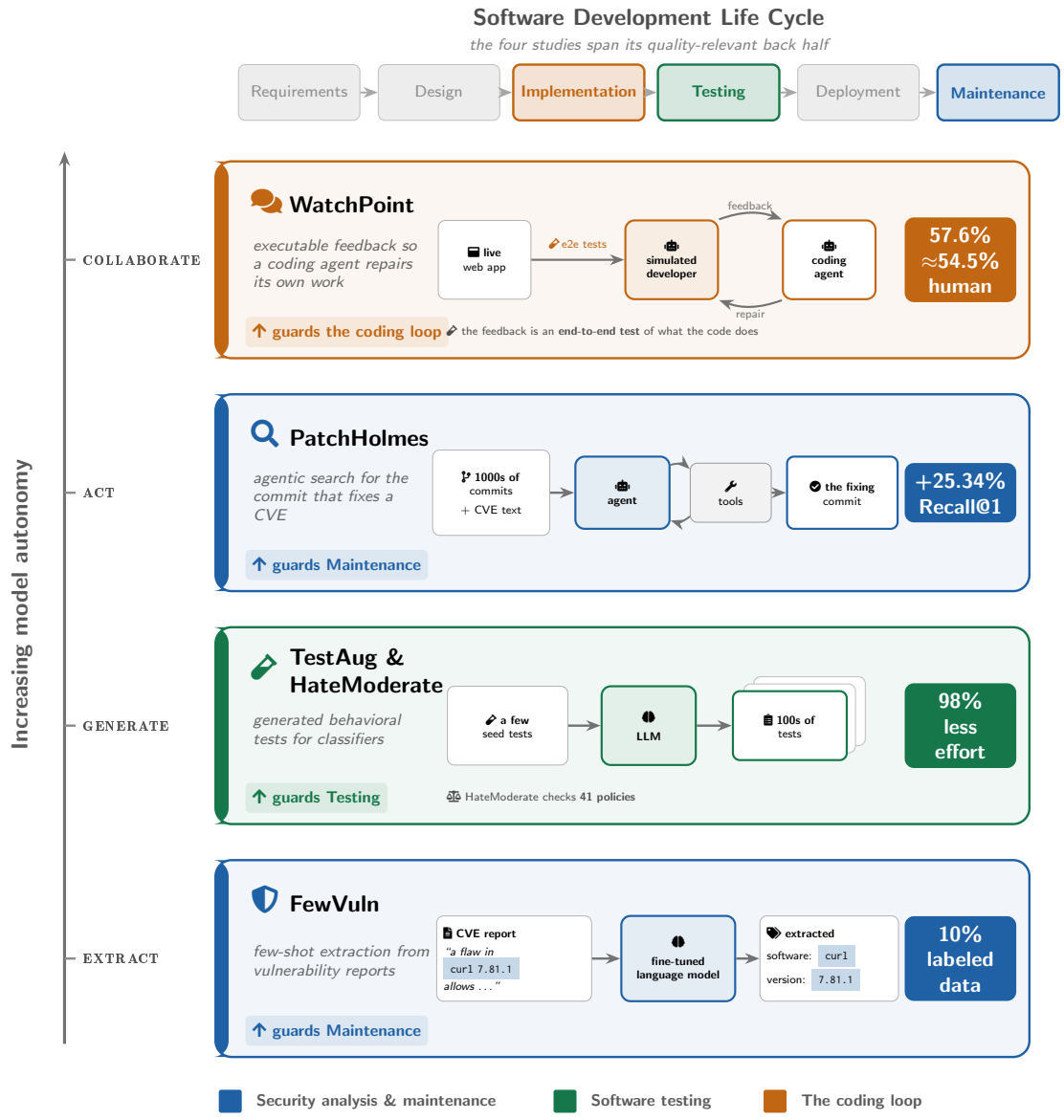


Figure 1.1: The four studies in this dissertation, placed on two axes. The vertical axis is autonomy: how much of the task the language model carries on its own, rising through EXTRACT (reporting facts), GENERATE (writing tests), ACT (using tools), and COLLABORATE (working with another agent). The horizontal axis is position in the software development life cycle: FewVuln and PatchHolmes manage vulnerabilities in existing software (security analysis and maintenance), TestAug and HateModerate generate behavioral tests (testing), and WatchPoint supplies executable feedback inside the coding loop, where AI agents now write software. Together the four studies span the quality-relevant back half of the life cycle. Each study's headline result appears beside it and is grounded in the corresponding chapter.

by hand.

- **Software analysis** secures a system by finding and managing its vulnerabilities: reading each disclosure, identifying the affected software, and locating the change that repairs it. A security analyst must read free-form reports, then search repositories that may hold millions of commits for the one fix that matters.

- **Software testing** checks that a system behaves as intended: writing the cases that probe its behavior and judging whether the outputs are acceptable. A test engineer must author probing examples one at a time and label the answer each should produce.

- **Testing inside the coding loop** is the newest front. AI coding assistants now write and revise a growing share of software (Liu et al., 2025), and their output cannot be trusted on sight: each change must be run and tested, and when a test fails, the result must be handed back so the assistant can repair it. This is testing moved from a final gate into the development loop, checking each AI-written change as it is produced.

As code, disclosed vulnerabilities, and AI-written software all grow faster than the supply of experts, the gap between what must be checked and who can check it widens. The one feature these activities share is that their raw material is language: a vulnerability report is a paragraph of English, a test case is a sentence paired with its expected answer, and a session with a coding assistant is a dialogue. This is why natural language processing (NLP), the branch of artificial intelligence that builds models of human language, is the natural tool; recent surveys document a shift across the field from merely detecting defects toward automatically generating and repairing them (Xu et al., 2025).

This Dissertation: Four Studies on Two Axes. This dissertation asks a single question: *Can NLP methods automate and scale the software testing and analysis that today require human expertise?* It answers with four studies, laid out in Figure 1.1 along two axes.

- **Position in the software life cycle** (the horizontal axis), following how recent work maps software-engineering tasks onto the stages of the development life cycle (Wang et al., 2026). The four studies cover its quality-critical back half: FewVuln and PatchHolmes manage vulnerabilities in existing software, TestAug and HateModerate test model behavior, and WatchPoint operates within the coding loop where new software is written.

- **Degree of model autonomy** (the vertical axis): how much of the task the language model carries on its own, and how little it needs a person beside it (Liu et al., 2025). Read in order, the studies climb four steps, from EXTRACT (FewVuln reads reports and pulls out facts), to GENERATE (TestAug and HateModerate write tests), to ACT (PatchHolmes uses tools on a repository), to COLLABORATE (WatchPoint works in a loop with another agent).

Software Analysis Is Maintenance-Stage Work. In this dissertation, *software analysis* names a family of techniques, information extraction and retrieval over security artifacts, while *maintenance* names a stage of the life cycle; the two coincide because security vulnerability management is post-release work. A vulnerability is disclosed against software already deployed in the field, so reading that disclosure to identify the affected software (FewVuln) and locating the commit that repairs it (PatchHolmes) are steps in remediating released software (Le et al., 2023; Dissanayake et al., 2021). Recent task surveys place this repair and patch work at the maintenance

stage, which they define as modifying deployed software to fix defects (Wang et al., 2026). The two analysis studies therefore sit at *maintenance* on the horizontal axis: analysis is their technique, and maintenance is where they act in the life cycle.

Headline Results and a Central Lesson. Across the four studies, applying a language model with care replaces manual work at scale: about 90% less annotation for extraction (FewVuln), over 98% less for test authoring (TestAug), a 25.34% gain in ranking the correct security patch first (PatchHolmes, in Recall@1, the share of vulnerabilities whose actual fix is the very first commit returned), and a 57.6% task-recovery rate that matches the 54.5% of human testers (WatchPoint). What the four studies share is that their leverage comes from how the language model is applied, through adaptation, prompting, tool use, and feedback. The two most autonomous studies go further and weigh this directly against the model's raw scale. PatchHolmes tested it head-on: replacing its model with one about seven times smaller changes the result by less than 1%, matching independent surveys of software-engineering agents that find more autonomy on its own does not guarantee better performance (Liu et al., 2025). WatchPoint sharpens rather than repeats the point: the value of a method can depend on the model's own ability, since its feedback helps a weaker coding model but can distract a stronger one.

1.1 Contributions

This dissertation makes four contributions, one per study, spanning the software life cycle from vulnerability management to testing to the coding loop, at four rising levels of autonomy.

Extracting Structured Facts from Security Vulnerability Reports. Chapter 3 presents FewVuln, a few-sample method for named entity recognition on

vulnerability reports. Named entity recognition is the task of marking the spans of a sentence that name an entity of interest, here a piece of software and its version. By fine-tuning a pre-trained language model, FewVuln matches prior work on software name and version extraction while using only about 10% of the training data, thereby reducing annotation effort by roughly 90% and making vulnerability tracking easier to scale as disclosures grow.

Generating Behavioral Tests for NLP Models. Chapter 4 presents TestAug and HateModerate. TestAug generates capability-based test cases with a large language model, reducing manual annotation by over 98% while uncovering more failures than human-written test suites. HateModerate extends the approach to content moderation by testing hate speech classifiers against Facebook's 41 community content policies and showing that fine-tuning on policy-aligned examples improves how closely a model adheres to the policies, without lowering its accuracy on ordinary inputs.

Acting on Code Repositories to Retrieve Vulnerability Patches. Chapter 5 presents PatchHolmes, an agent that finds the commit fixing a given vulnerability by inspecting a list of candidates through a small set of budgeted tools. It improves Recall@1 by 25.34% over a strong baseline that scores each candidate on its own, at the cost of one agent conversation per vulnerability, and it shows that the structure built around the model, not the size of the model, drives the result: replacing the model with one seven times smaller changes Recall@1 by under 1%.

Collaborating with Coding Agents through Executable Feedback. Chapter 6 presents WatchPoint, a simulated-developer system that helps an AI coding agent repair its own work by running diagnostic scripts against the live application and returning what it observes, rather than grading a stored account of what it did. It recovers 57.6% of the failing tasks it inspects, and a controlled user study finds

this to be close to the 54.5% recovered by human testers, validating the simulated developer as a realistic and reproducible stand-in for interactive human feedback.

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows.

- **Chapter 2** provides background on the NLP methods used throughout the dissertation, on security vulnerability reports and their patches, and on software testing, with attention to the special difficulties of testing machine learning systems and the agents built on top of language models.
- **Chapter 3** presents few-sample named entity recognition for security vulnerability reports, the EXTRACT stage of the progression.
- **Chapter 4** presents TestAug and HateModerate, automated test generation for language models, the GENERATE stage.
- **Chapter 5** presents PatchHolmes, agentic patch retrieval for vulnerability management, the ACT stage.
- **Chapter 6** presents WatchPoint, executable user feedback for coding agents, the COLLABORATE stage.
- **Chapter 7** summarizes the contributions and discusses their broader implications.
- The **Appendix** contains supplementary materials, including detailed dataset statistics, additional experimental results, and the exact provider identifiers for every model used.

Taken together, these four studies make one argument: NLP methods, applied with care, can take over software testing and analysis across a wide range, from reading a security report to working alongside the coding agents that now help build software. The chapters that follow develop each study in turn, and the rising autonomy of Figure 1.1 is the thread that connects them.

Chapter 2

Background

This chapter provides the background for the four studies in this dissertation. Chapter 1 placed those studies on two axes: the software development life cycle and a ladder of increasing autonomy (Figure 1.1). Of its stages, the field of automated software engineering has invested most heavily in *implementation*, that is, in writing code. A recent survey of code-model benchmarks finds that implementation accounts for the majority of work, while testing and maintenance lag far behind (Wang et al., 2026). These two underserved stages are the focus of this dissertation because they determine a system's correctness and security: *software testing*, which checks that a system behaves as intended, and *software analysis*, which secures a system by managing its vulnerabilities. The chapter is organized to mirror the dissertation's title. It first covers software testing, the domain of TestAug, HateModerate, and WatchPoint. It then covers software analysis, the domain of FewVuln and PatchHolmes. It closes with the NLP methods that make both scalable.

2.1 Software Testing

Software testing aims to find defects and to confirm that software behaves as expected. Despite decades of work, it remains one of the most labor-intensive parts of development, often consuming 30-50% of the total effort (Myers, 2012). This section follows software testing along a single continuum, shown in Figure 2.1, from a final gate applied to a finished artifact to an interactive signal produced inside the development loop. Reading that figure from left to right, the section covers traditional testing, the new challenges introduced by machine learning systems, capability-based

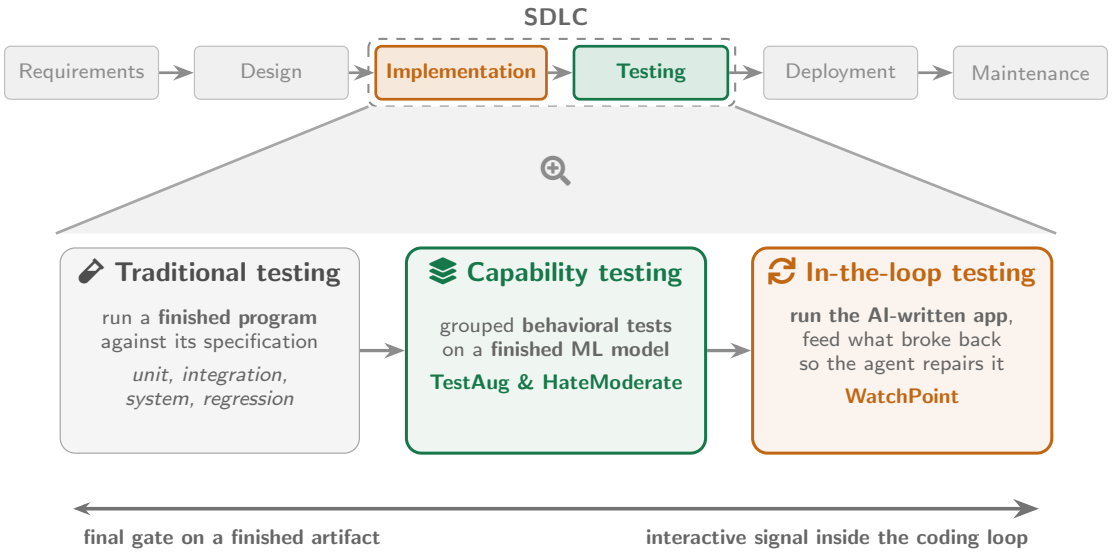


Figure 2.1: Software testing along a single continuum, unboxing the testing and implementation stages of the software development life cycle from Figure 1.1. Testing runs from traditional testing of a finished program, to capability-based testing of a finished machine learning model (TestAug and HateModerate), to executable testing inside the coding loop where an AI agent writes the software (WatchPoint), moving from a final gate on a finished artifact to an interactive signal inside the development loop.

testing that addresses them, and testing AI coding assistants in the loop. These are the domains of TestAug, HateModerate, and WatchPoint; the first two generate behavioral tests for a finished model, and the third moves testing into the coding loop where AI now writes software (Figure 1.1).

2.1.1 Traditional Software Testing

Software testing is "the process of executing a program with the intent of finding errors" (Myers, 2012). This framing treats testing as a search for defects rather than a demonstration of correctness, so a good test case is one that has a high chance of revealing an error. As Dijkstra (1972) famously observed, "Program testing can be used to show the presence of bugs, but never to show their absence." Because no

finite set of tests can prove a program correct, testing must be systematic.

Traditional testing uses several complementary strategies (Myers, 2012; Crispin and Gregory, 2014):

- **Unit testing** checks individual functions in isolation, specifying inputs and the outputs they should produce.
- **Integration testing** checks that components work together correctly across their interfaces.
- **System testing** checks the complete system against its specification, from the user's point of view.
- **Regression testing** re-runs tests after a change to confirm that existing behavior still holds.

To make the idea of a test case concrete, consider the function in Listing 2.1, which classifies a triangle by its side lengths:

Listing 2.1: A triangle-classification function, used as a running example of how even a small program needs careful test design.

```
def classify_triangle(a, b, c):
    """Classify a triangle by its side lengths."""
    if a <= 0 or b <= 0 or c <= 0:
        return "invalid"
    if a + b <= c or b + c <= a or a + c <= b:
        return "invalid"
    if a == b == c:
        return "equilateral"
    if a == b or b == c or a == c:
        return "isosceles"
    return "scalene"
```

A thorough set of tests should cover valid triangles of each type, invalid inputs such as zero or negative sides, boundary cases where the sides just fail the triangle

inequality, and different orderings of the same sides. Even this small function requires careful test design to ensure adequate coverage (Myers, 2012).

Two broad strategies guide the selection of test cases (Myers, 2012). **Black-box testing**, also called behavioral testing, derives tests from the specification without looking at the code, focusing on the relationship between inputs and outputs. **White-box testing**, also called structural testing, derives tests from the code itself, aiming to exercise its statements, branches, and paths. Effective testing combines the two: black-box testing checks that the program meets its specification, and white-box testing checks that the tests actually exercise the code.

2.1.2 Testing Machine Learning Systems

Machine learning systems break the assumption that traditional testing rests on (Huyen, 2022; Kästner, 2025). In ordinary software, a developer writes the rules, so there is a specification to test against. In a machine learning system, the model learns its behavior from data, so there is no written rule that says what the output should be. This gap has created an active research area focused on testing machine learning systems (Zhang et al., 2022).

The missing specification is the core of the problem. For the triangle classifier, we can state that the input $(3, 4, 5)$ must be classified as "scalene". For a sentiment classifier, we cannot list every sentence and its correct label because the mapping is too complex to write down (Kästner, 2025). As a result, we cannot prove a model correct; we can only check that it does well on the data we have (Zhang et al., 2022). Deciding the correct output for an arbitrary input, known as the *oracle problem*, may itself require human judgment.

The usual way to evaluate a model is a held-out test set, a portion of the data set aside to estimate performance on unseen examples. This gives one aggregate

number, and that number hides a great deal (Geva et al., 2019; Gururangan et al., 2018):

- **Aggregate scores mask failures.** A model that is 90% accurate overall can be wrong almost every time on an important subset of inputs.
- **Test sets can be biased.** They may not match real-world data, and spurious correlations between labels and surface features can let a model score well without learning the task (Gururangan et al., 2018).
- **Scores are not interpretable.** An accuracy number does not say which capabilities the model has or lacks, so it gives little guidance for improvement.
- **Benchmarks can be overfit.** Repeated evaluation on the same benchmark can reward fitting its quirks rather than the underlying skill (Geva et al., 2019).

For example, a natural language inference model can reach 90% accuracy while failing almost every time the sentence contains a negation, a pattern that a single accuracy number would never reveal. This is what motivates more fine-grained testing.

2.1.3 Capability-Based Testing for Machine Learning Models

To overcome the limitations of held-out evaluation, researchers proposed *behavioral* or *capability-based* testing, which evaluates a model on specific capabilities rather than a single aggregate score (Ribeiro et al., 2020; Yang et al., 2022). The idea borrows from behavioral testing in software engineering and adapts it to machine learning, where code-coverage metrics do not apply.

Capability-based testing groups test cases by the skill they probe. To test whether a sentiment classifier understands negation, for instance, the cases might include "I love this movie" (positive), "I do not love this movie" (negative), "No one

likes this movie" (negative), and "This movie is not bad" (positive, a double negative). Grouping tests this way lets a developer see exactly which skill is weak and track it over time. The CheckList framework (Ribeiro et al., 2020) organizes this into reusable test types: minimum functionality tests that check a basic skill, invariance tests where a label-preserving change should not alter the prediction, and directional tests where a change should move the prediction in a known direction. Later work extended the idea to specific domains such as hate speech detection (Röttger et al., 2021) and question answering (Ribeiro and Lundberg, 2022). The difficulty is that building a thorough capability-based suite by hand takes substantial effort and expertise, which is the bottleneck Chapter 4 removes by generating the tests automatically.

Capability-based testing remains a final gate: it probes a finished model using a fixed set of cases. The next subsection follows testing one step further, to software that an AI agent is still writing.

2.1.4 Testing AI Coding Assistants in the Loop

Software is increasingly written by AI coding assistants, and that software has to be tested like any other. Testing it, however, differs from the settings above in two ways: the artifact under test is produced by an agent rather than a person, and the assistant does not return a fixed output but holds a conversation, producing code, reacting to what breaks, and revising. Testing such an assistant means testing this whole exchange, not a single response.

Testing AI-written code is now a recognized activity. The community has begun to test the projects that coding agents build end-to-end, compiling and running them against comprehensive suites and returning fine-grained failure verdicts rather than a single pass-or-fail bit (Lu et al., 2026). This sits within the broader turn toward using language models throughout software testing: a survey of that literature

organizes it around recognized testing tasks, including system-level test input generation and test-feedback-driven debugging and repair, and documents agents that drive a running application to test it, keeping the loop of action and observation going (Wang et al., 2024).

Driving the system is the key idea. For an interactive application, there is no fixed input-output oracle, so the tester must exercise the system as a user would. Autonomous agents now do this: in automated web GUI testing, LLM agents act as testers that click, type, and fill forms on the live application under test, using code coverage and runtime errors as the oracle, and looping execution results back to plan further tests (Liu et al., 2025). Simulating the user is therefore not a soft substitute for evaluation but the mechanism by which interactive software is tested: simulation supplies the interaction, and execution supplies the oracle (Balog and Zhai, 2025; Park et al., 2023; Lu et al., 2025; Feng et al., 2025).

Most current benchmarks for coding assistants do not go this far. They treat the user as fixed, supplying a frozen prompt or a stored test script and scoring the assistant once against it (Jimenez et al., 2023; Xu et al., 2025). This makes results repeatable, but it leaves out the interaction that defines real use. Recent work directly shows the gap: when static benchmark questions are turned into user-AI conversations, a model's accuracy on its own does not predict its accuracy when a user is in the loop (Chang et al., 2025). An assistant that looks strong in isolation can be weak at handling feedback, and a static benchmark will not notice. What is missing is a tester that does what a developer does: open the running application, exercise it, read what went wrong, and report it back.

Chapter 6 builds exactly this tester. WatchPoint is a simulated developer that, instead of judging text, writes and runs diagnostic scripts against the live application and returns the raw observations to the coding assistant it works with, so that

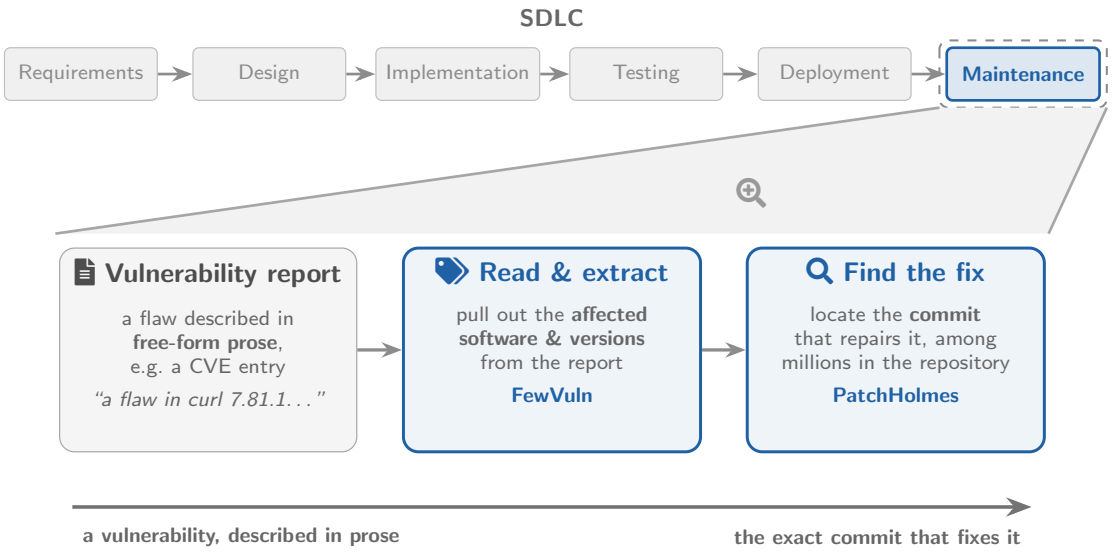


Figure 2.2: Software analysis as security vulnerability management, unboxing the maintenance stage of the software development life cycle from Figure 1.1. A vulnerability described in free-form prose is read so that the affected software and versions are extracted (FewVuln), and then the commit that repairs it is located among the millions in the repository (PatchHolmes).

assistant can repair its own work. It is evaluated on a benchmark of multi-file web projects whose tasks depend on one another and are checked by automatic end-to-end tests (Xu et al., 2025). This is the kind of realistic, execution-grounded setting that recent agent benchmarks have moved toward (Zhou et al., 2024; Wang et al., 2025). A controlled study with human testers confirms that this executable feedback recovers failing tasks at close to the human rate. WatchPoint thus completes the testing continuum of this section: from behavioral tests applied to a finished model (TestAug, HateModerate) to executable tests produced inside the coding loop, where testing has moved to keep pace with software that AI now writes (Figure 1.1).

2.2 Software Analysis

The previous section covered software testing; this section turns to the dissertation's other quality activity, software analysis. Software systems produce a large body of text, including bug reports, commit messages, documentation, and security advisories, that records how the system behaves and where it fails. Applying NLP to these artifacts has become an active research area (Yang et al., 2022; Fan et al., 2023). In this dissertation, software analysis means *security vulnerability management*, the domain of FewVuln and PatchHolmes, which sits at the maintenance end of the life cycle (Figure 1.1). The two studies form a short sequence, shown in Figure 2.2: a vulnerability report is read so that its affected software and versions are extracted (FewVuln), and then the commit that fixes the vulnerability is found among the millions in the repository (PatchHolmes). This section first surveys software analysis tasks, then describes vulnerability reports and the challenge of reading them, and finally describes the separate challenge of linking a vulnerability to the commit that fixes it.

2.2.1 Software Analysis Tasks

Deep learning has been applied across the software lifecycle, from requirements to maintenance (Yang et al., 2022). The common thread is that textual software artifacts contain information that can be automatically extracted and used. Representative tasks include:

- **Requirements engineering**: extracting features, actors, and actions from natural language requirements to support traceability and consistency checking (Guo et al., 2017).
- **Code comprehension**: generating summaries, documentation, and explana-

tions that help developers understand unfamiliar code (Yang et al., 2022; Fan et al., 2023).

- **Defect prediction and bug localization**: pointing to the likely locations of defects by learning from historical bug data (Yang et al., 2022; Li et al., 2022).
- **Software maintenance**: supporting evolution through clone detection, change recommendation, and review classification (Yang et al., 2022).
- **Vulnerability assessment**: estimating the exploitability, impact, and severity of vulnerabilities to help triage them (Le et al., 2023).

The scale of modern development makes such automation necessary. Public vulnerability databases alone receive tens of thousands of new entries each year (Le et al., 2023; Croft et al., 2023), far more than any team can process by hand. FewVuln and PatchHolmes address two of these tasks in turn: reading a vulnerability report, and finding its fix.

2.2.2 Security Vulnerability Reports

Security vulnerabilities can let attackers steal data, disrupt services, or gain unauthorized access (Gaglione Jr, 2019), and high-profile incidents have shown that a single flaw can affect millions of systems and cause large financial losses (Le et al., 2023). To manage vulnerabilities systematically, the security community maintains public databases. Two are central:

- **Common Vulnerabilities and Exposures (CVE)**, maintained by MITRE Corporation, gives each publicly known vulnerability a standard identifier (Farhang et al., 2020). An entry includes the identifier, a natural language description, the affected software, and references to patches or advisories. The

list has archived over 100,000 entries and is the de facto standard for naming vulnerabilities (Dong et al., 2019).

- **National Vulnerability Database (NVD)**, maintained by the U.S. National Institute of Standards and Technology, adds severity scores through the Common Vulnerability Scoring System and weakness categories through the Common Weakness Enumeration (Dong et al., 2019). NVD analysts convert the free-form CVE text into structured fields, such as the affected software names and versions, that automated tools can use (Dong et al., 2019).

These databases let security teams track vulnerabilities, prioritize patching, and assess exposure. The number of reports continues to grow, with more than 20,000 new entries in recent years (Le et al., 2023; Croft et al., 2023), straining manual processing and motivating the development of automated tools.

2.2.3 Information Extraction from Vulnerability Reports

A basic obstacle is that the most useful information in a report is written in prose rather than stored in structured fields (Dong et al., 2019). A description might say "Internet Explorer 11 on Windows 7" or "Apache HTTP Server 2.4.0 through 2.4.49" without marking these as machine-readable software names and versions (Yitagesu et al., 2023). A security analyst must read the text and extract these facts before deciding what to do, a process that requires time and security expertise (Yitagesu et al., 2023; Bridges et al., 2013).

Extracting this structure enables several downstream uses:

- **Asset matching**: deciding which vulnerabilities affect deployed software by matching extracted names and versions against an inventory (Yitagesu et al., 2023).

- **Inconsistency detection**: spotting disagreements between databases. Only about 60% of reports match the standardized NVD entry exactly, and the rest can leave systems unpatched (Dong et al., 2019).
- **Knowledge base construction**: building security knowledge graphs that link related vulnerabilities and products (Yitagesu et al., 2023; Gao et al., 2021).
- **Threat intelligence**: correlating extracted entities with exploit databases to judge active risk (Le et al., 2023; Gao et al., 2021).

Named entity recognition is the core technique for this task (Bridges et al., 2013; Gasmi et al., 2019). Given a description, the system marks the spans that name software, versions, and other entities. The setting is hard for several reasons: software names and versions are extremely diverse, so dictionaries do not cover them; descriptions mix prose with code and domain-specific phrasing that confuse standard tools; and the system must tell vulnerable software apart from software that is merely mentioned (Dong et al., 2019; Yang et al., 2022).

The greater difficulty is a shortage of labeled data. Large annotated datasets for vulnerability extraction are scarce because labeling requires security expertise, which is itself in short supply (Yitagesu et al., 2023; Croft et al., 2023). Models trained only on general text lose accuracy on security text (Yitagesu et al., 2023), and the wording of descriptions varies enough that rule-based methods cannot keep up (Dong et al., 2019; Bridges et al., 2013). This shortage motivates *few-sample* methods that aim to match full-data accuracy from a small labeled set. Prior work has approached few-shot named entity recognition through metric learning (Snell et al., 2017), meta-learning (Li et al., 2020), and prototype methods (Fritzler et al., 2019), and pre-trained language models have shown strong few-shot performance when carefully fine-tuned (Dodge et al., 2020; Huang et al., 2020). Chapter 3 follows

this last line: fine-tuning a pre-trained model reaches state-of-the-art extraction on vulnerability reports using only about 10% of the original training data.

2.2.4 Linking Vulnerabilities to Fixes

Reading a report is only the first half of vulnerability management. The second half is finding the change that actually fixes the vulnerability, that is, the specific commit in the software's source repository. This link is essential for downstream work: it tells a security team which versions are affected, lets them verify and back-port the fix, and feeds the supply-chain scanners that warn other projects about vulnerable dependencies (Le et al., 2023; Bhandari et al., 2021). Yet most vulnerability reports never record this link, so it must be recovered after the fact. Chapter 5 reports that the large majority of entries in major advisory databases lack a patch link, which makes automatic recovery a problem worth solving rather than a rare exception.

The task, called *patch retrieval*, is to find the fixing commit among all the commits in a repository. It is hard for three reasons that Chapter 5 examines in detail. First, repositories are large, often holding thousands to millions of commits, so the fix is one commit hidden among a great many. Second, the report and the commit describe the same flaw in different words: the report names the security symptom, while the commit message names the underlying code change, so they share few words to match on. Third, the change itself can be long and span several files, which strains methods that read only short snippets.

Prior approaches typically score each candidate commit on its own and then sort by the score, a strategy known as *pointwise* ranking (Chapter 5). The weakness of scoring candidates one at a time is that the method never compares them directly, so when several commits look plausible, it has little basis for choosing among them. PatchHolmes takes a different approach, built on the language-model agents intro-

duced in the next section. It pairs a fast first stage that narrows the repository to a short list of candidates with a second stage in which an agent reads the whole list at once and inspects a few commits through budgeted tools before naming the fix. Reading the candidates together, a strategy known as *listwise* selection, lets the agent weigh them against one another rather than in isolation. Chapter 5 shows that this agentic, listwise design substantially outperforms pointwise baselines, and that the agent scaffolding, not the size of the language model, is what drives the gain.

2.3 NLP Methods for Scalable Software Testing and Analysis

The two sections above described what software testing and analysis require and why manual effort cannot keep up. This section introduces the NLP methods this dissertation builds on to close that gap. Natural language processing has shifted in recent years from building a separate model for each task to adapting one large pre-trained model to many tasks (Qiu et al., 2020). Four such methods run through the studies that follow, one for each rising level of autonomy in Figure 1.1: fine-tuning supports EXTRACT, in-context learning supports GENERATE, tool-using agents support ACT, and language-model-based simulation supports COLLABORATE. The section introduces the four methods in turn and then explains why, together, they turn bounded human effort into coverage that scales.

2.3.1 Fine-Tuning Pre-trained Language Models

Pre-trained language models such as BERT (Devlin et al., 2019), RoBERTa (Liu et al., 2019), and GPT (Radford and Narasimhan, 2018) changed how NLP systems are built (Rogers et al., 2020; Qiu et al., 2020). Instead of training a model from scratch on task-specific data, the modern recipe has two stages. First, the model is

pre-trained on a massive unlabeled corpus to learn general representations of language. Second, it is fine-tuned on a smaller labeled dataset for a specific task. This recipe improved results across nearly every benchmark and made pre-trained models the default starting point for new applications (Qiu et al., 2020).

These models are built on the Transformer architecture (Vaswani et al., 2017), whose self-attention mechanism captures long-range relationships in text more effectively than the recurrent networks that came before it (Li et al., 2022). Pre-training uses objectives that need no human labels. BERT uses masked language modeling, hiding a fraction of the tokens (typically 15%) and learning to predict them from the surrounding context on both sides. GPT-style models use next-token prediction, predicting each token from the tokens before it, which makes them natural text generators. Through these objectives, the model absorbs grammar, meaning, and a great deal of factual knowledge from billions of words (Rogers et al., 2020).

Fine-tuning then adapts this knowledge to a task. For sequence labeling tasks such as named entity recognition, a small classification layer is added on top of the model to predict a label for each token. For text classification, the layer predicts a single label for the whole input. The key point is that pre-training already supplies most of the language understanding, so fine-tuning needs far less labeled data than training from scratch (Rogers et al., 2020). This advantage is largest when labeled data is scarce, which is exactly the setting of security vulnerability analysis. As Chapter 3 shows, fine-tuning a pre-trained model reaches state-of-the-art extraction of software names and versions using only a small fraction of the training data that prior methods required.

2.3.2 In-Context Learning

Fine-tuning updates the model's parameters. Large language models such as GPT-3 support a second way to adapt, called *in-context learning*, that changes no parameters at all (Brown et al., 2020). Instead, the model is given a prompt that describes the task and, optionally, shows a few worked examples, and it then produces the answer for a new input. This ability appears mainly in large models and grows in a predictable way as the model grows (Kaplan et al., 2020; Hoffmann et al., 2022). The broader framing, sometimes called "pre-train, prompt, and predict", treats prompting as a genuine alternative to fine-tuning (Liu et al., 2021).

In-context learning covers a range of prompting strategies of increasing sophistication:

- **Zero-shot prompting** gives the model only a task description, with no examples, for instance "Classify the following text as positive or negative sentiment". Performance rests entirely on what the model learned during pre-training and on its ability to follow instructions (Reynolds and McDonell, 2021).

- **Few-shot prompting** adds a small number of input-output examples (often 1 to 32) before the test input. The model infers the pattern from these examples without any parameter update, though the choice and ordering of the examples matter (Gao et al., 2021).

- **Chain-of-thought prompting** includes worked reasoning steps in the examples, prompting the model to reason step by step before answering. This helps most on tasks such as arithmetic and commonsense reasoning (Wei et al., 2023; Wang et al., 2023).

Instruction-tuned models, which are fine-tuned on instructions paired with

desired answers and then refined with reinforcement learning from human feedback, follow natural language prompts more reliably and make in-context learning more effective (Ouyang et al., 2022). In-context learning is attractive when labeled data is scarce or when tasks change frequently, though it can be less data-efficient than fine-tuning for a fixed task with ample labels. As Chapter 4 shows, in-context learning is well suited to generating test cases, because the model's broad knowledge lets it produce varied examples, including edge cases that a human author might overlook.

2.3.3 Tool-Using Language Model Agents

The two methods above use a language model as a single function: text goes in, text comes out. A third method places the model inside a loop and lets it take actions in an environment. A model used this way is called an *agent*. At each step, the agent reads its current situation, chooses an action, executes it via a tool, observes the result, and decides what to do next. The loop repeats until the agent reports an answer or gives up. This turns a passive predictor into something that can act on the world, which is what the higher levels of autonomy require.

An agent is usually described in terms of a few components (He et al., 2025). *Planning* breaks a goal into steps and revises them as new information arrives. *Tools* are the actions the agent can take, such as reading a file, running a command, or querying a search index, each exposed to the model as a function it may call. *Memory* records what the agent has seen and done so it can use that history later. *Observation* is how results flow back into the model for the next decision. The model supplies the reasoning; the surrounding components, often called the *scaffolding*, supply the structure.

Agents have become a leading approach to software engineering tasks that a single prompt cannot solve. On the SWE-bench benchmark, agents attempt to resolve

real software issues by editing a repository and running its tests (Jimenez et al., 2023), and systems built for this setting range from lightweight, fixed procedures (Xia et al., 2024) to general open platforms for software agents (Wang et al., 2025,?). Beyond code, agents act in web environments where they click, type, and read pages to complete tasks (Zhou et al., 2024; Koh et al., 2024). A recurring lesson across this work is that the scaffolding around the model, the tools it is given and the way its decisions are structured, can matter as much as the choice of model. Chapter 5 reports a sharp example of this lesson: in a patch-retrieval agent, the scaffolding accounts for most of the accuracy, while replacing the language model with a much smaller one barely changes the result. Chapters 5 and 6 build directly on the agent paradigm, the first to act on a code repository and the second to drive a multi-agent loop.

2.3.4 Language-Model-Based Simulation and Evaluation

Interactive systems such as chatbots, dialogue agents, and coding assistants are hard to test with traditional benchmarks (Zheng et al., 2023). They work in open-ended settings where there may be no single correct answer, and their quality depends on how a user reacts rather than on a fixed key. Two language-model-based approaches address this difficulty.

User simulation replaces the human user with a model that imitates how a user behaves, which makes testing both repeatable and cheap (Balog and Zhai, 2025; Zhang et al., 2017). Controlled comparison across systems is only possible when the user behaves consistently each time, and a real person does not (Zhang et al., 2017). Large language models make these simulated users far more capable than the rule-based simulators of the past: they produce natural language, adapt to new situations, and reproduce subtle aspects of behavior that are hard to write down by hand (Park et al., 2023; Zhang et al., 2024; Philipov et al., 2024). This is especially useful for

assistants, where what the user says next depends on what the assistant just did, and it is the mechanism behind the in-the-loop testing of Chapter 6.

Language-model-as-a-judge uses a strong model to score the open-ended outputs of another model, where metrics such as BLEU or accuracy do not apply (Zheng et al., 2023; Wagner et al., 2024). The judge can compare two answers, grade a single answer, or check an answer against a reference. Strong judges agree with human raters at rates above 80%, close to how often two humans agree (Zheng et al., 2023), though they carry known biases, such as favoring the first answer they see or the longer one, which motivates ongoing work on correcting them (Wagner et al., 2024).

Both approaches judge text. For a coding assistant, however, the most informative feedback is not a score on the assistant's own account of its work but the result of running the software. Chapter 6 builds on user simulation but moves past text-level judgment: its simulated developer writes and runs diagnostic scripts against the live application and reports what it observes, giving the assistant the kind of executable feedback a real developer would provide.

2.3.5 Why NLP Methods Make Testing and Analysis Scalable

The four methods above share one effect, as shown in Figure 2.3: each takes a small, bounded amount of human effort and amplifies it into coverage that grows with the software, rather than with the number of available experts. This is what makes them a scalable solution to the testing and analysis problems in the previous two sections, whose common obstacle was that expert human effort does not keep pace with the growth of code, vulnerabilities, and AI-written software.

The amplification works because the raw material of both testing and analysis is language, and each method turns one linguistic act into many:

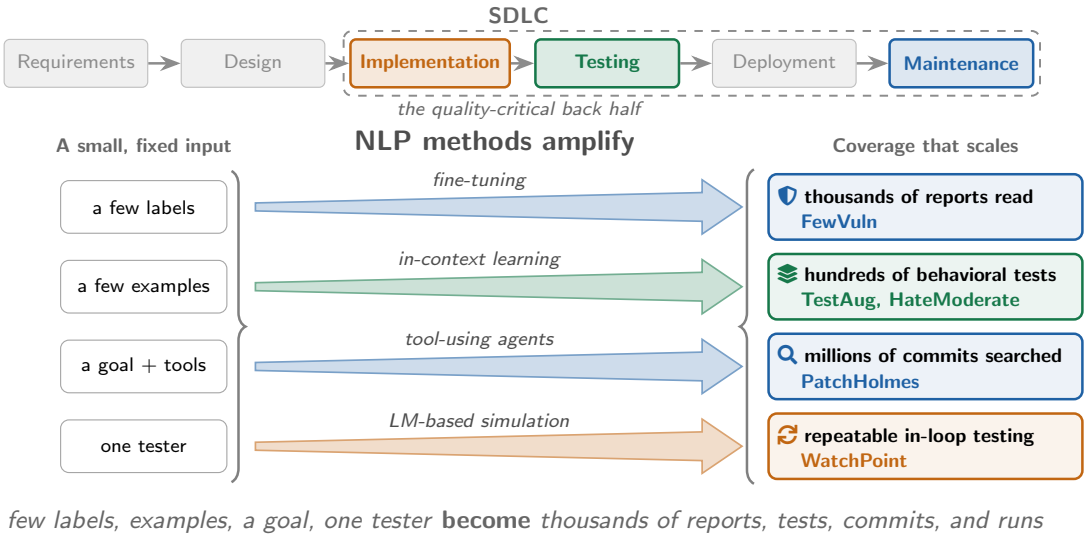


Figure 2.3: How NLP methods make testing and analysis scalable. Each method amplifies a small, fixed human input into coverage that grows with the software: a few labeled examples, a few seed tests, a goal with a few tools, and one simulated tester become, respectively, thousands of vulnerability reports read (FewVuln), hundreds of behavioral tests (TestAug and HateModerate), millions of commits searched (PatchHolmes), and repeatable in-the-loop testing (WatchPoint), across the quality-critical back half of the software development life cycle (Figure 1.1).

- **Fine-tuning** amplifies a handful of labeled examples. A security analyst labels a small set of vulnerability reports, and the fine-tuned model then reads the rest, extracting software names and versions across a stream of reports no team could process by hand (FewVuln).

- **In-context learning** amplifies a handful of seed tests. A few example behaviors become hundreds of generated test cases that probe a model's capabilities, including edge cases a human author would miss (TestAug, HateModerate).

- **Tool-using agents** amplify a single goal. Given the goal of finding a fix and a few read-only tools, an agent searches a repository of millions of commits on its own, work that would otherwise fall to a human analyst (PatchHolmes).

- **Language-model-based simulation** amplifies a single tester. One simulated developer runs the application and reports what broke, serving as a proxy for the costly, slow, and inconsistent human studies that in-the-loop testing would otherwise require (WatchPoint).

Read together with Figure 1.1, the pattern is that the same small human input, a few labels, a few examples, a goal, one tester, is amplified further as autonomy rises, through EXTRACT (perceiving software artifacts), GENERATE (generating tests), ACT (acting on a codebase), and COLLABORATE (testing inside the coding loop). The methods do not replace human judgment; they extend its reach so that testing and analysis can scale to match the pace at which software and the agents that now help write it are produced.

2.4 Summary

This chapter introduced the foundations for the four studies that follow, organized to mirror the dissertation's title. It first surveyed software testing, from traditional methods, through the special difficulties of testing machine learning systems, to testing AI coding assistants in the loop. The missing specification and the limits of held-out evaluation motivate the capability-based test generation of Chapter 4, and the interactive nature of AI-written software motivates the executable, in-the-loop testing of Chapter 6.

It then examined software analysis, in the form of security vulnerability management, the domain of FewVuln and PatchHolmes. Reading a vulnerability report is an extraction problem made hard by unstructured text, domain-specific language, and a shortage of labeled data, which motivates the few-sample method of Chapter 3. Finding the commit that fixes a vulnerability is a separate retrieval problem made

harder by repository size and the wording gap between a report and its fix, which motivates the agentic, listwise method of Chapter 5.

Finally, it presented the four NLP methods that power these studies: fine-tuning pre-trained models, which gives strong accuracy from little labeled data; in-context learning, which adapts a model through prompting rather than parameter updates; tool-using agents, which let a model act on software through a loop of decisions and observations; and language-model-based simulation and evaluation, which makes the testing of interactive systems repeatable. Together, these methods amplify a small amount of human effort into coverage that scales, which is what lets the chapters that follow take on testing and analysis at the pace modern software demands.

Chapter 3

Extracting Structured Facts from Security Vulnerability Reports

This chapter begins the two-axis arc of Chapter 1 at its first step on both axes (Figure 1.1). On the software development life cycle, it addresses *vulnerability management*, the analysis of software after release; on the axis of autonomy, it takes the first step, EXTRACT, where the language model perceives a software artifact and reports structured facts about it while a human still defines the task. The task is named entity recognition for security vulnerability reports, and the technique is fine-tuning a pre-trained language model (Chapter 2). We present FewVuln, which matches prior work on extracting software names and versions while labeling only a small fraction of the usual data. The bottleneck it removes is manual annotation; each subsequent chapter climbs one step higher.

3.1 Introduction

Security vulnerabilities pose great challenges to software and network systems. Numerous reported data breaches at major companies have jeopardized the information security of hundreds of millions of customers (Gaglione Jr, 2019); vulnerable automation software has been implicated in critical system failures (Palmer, 2020). To allow tracking and management of security vulnerabilities, organizations have created public security vulnerability reports stored in databases such as the Common Vulnerabilities and Exposures (CVE) database maintained by MITRE and the National Vulnerability Database (NVD) maintained by the U.S. government. These databases have largely helped security companies and vendors manage and prioritize patch deployment for their customers.

To scale up vulnerability management, existing work has leveraged NLP techniques to extract information from unstructured vulnerability reports, including software name, software version, and vulnerability type. The extraction of such information can help accelerate security administration in several scenarios: First, after converting unstructured reports into structured entries, such as software name and version, this information can be used directly to detect inconsistencies across vulnerability reports. Significant inconsistencies have been detected between reports from the CVE and NVD databases (Dong et al., 2019), which may cause system administrators to retrieve outdated or incorrect security alerts; inconsistencies were also detected between two entries created by the same vendor (Farhang et al., 2020). Second, extracted entries can be used in downstream applications, such as constructing knowledge ontologies to model relationships among security entries (Gao et al., 2021; Bridges et al., 2013) and automatically generating natural-language summaries of reports (Sun et al., 2021).

The scalable processing of vulnerability reports into structured formats relies on an effective information extraction (IE) system. With massive amounts of reports, manual extraction of information is intractable, and most existing systems rely on machine learning approaches. For example, Mulwad et al. (2011) build an SVM-based system to extract security concepts from web text. More recent systems apply deep learning methods, such as LSTM (Gasmi et al., 2019; Zhou et al., 2020) and bidirectional GRU (Dong et al., 2019). Deep learning has been shown to be effective in this task, achieving F1 scores as high as 98% (Dong et al., 2019).

Nevertheless, a major challenge of applying existing systems (e.g., recurrent neural networks like GRU (Dong et al., 2019) and LSTM (Gasmi et al., 2019; Zhou et al., 2020)) is that they often require a large number of labeled training samples to perform well; this number could range from a few thousand (Zhang et al., 2020)

to tens of thousands (Gasmi et al., 2019). Labeling a NER dataset specific to the computer security domain at this scale is both costly and time-consuming. This problem is further exacerbated by the fact that new security reports are routinely generated. Therefore, it is vital to investigate opportunities to alleviate the burden of labeling large corpora, i.e., few-sample learning.

In this chapter, we investigate, for the first time, the problem of information extraction from public vulnerability reports in a setting where only a small number of labeled training samples are available. Our goal is to minimize the number of labeled samples while avoiding significant performance degradation.

In recent years, significant progress has been made in few-shot learning for tasks such as image classification (Finn et al., 2017), text classification (Gao et al., 2021), and text generation (Brown et al., 2020). In these tasks, the training set often contains 10 or fewer examples, and the available examples are sufficient to achieve good performance (Chang and Chen, 2018). To generalize to unseen examples, few-shot learning approaches often leverage meta-learning (Finn et al., 2017), metric learning (Snell et al., 2017; Chen et al., 2021), and model pre-training. For example, the GPT-3 model (Brown et al., 2020) contains 175 billion parameters and was pre-trained on 45TB of unlabeled text data. For information extraction, specifically named entity recognition (NER), several existing works examine the few-shot setting in which only tens of labeled samples are available during training (Yang and Katiyar, 2020; Hou et al., 2020). However, NER systems to date can only achieve around 62% F1 score on general-domain datasets such as CoNLL 2003 (Tjong Kim Sang and De Meulder, 2003). When applied to security vulnerability reports, it is unclear whether performance can be satisfactory for security administrators and vendors (Dong et al., 2019).

We investigate the following research questions:

- Is it possible to match the state-of-the-art performance in NER for security vulnerability reports by using only a small number of labeled examples?
- What is the minimum number of labeled examples required for this problem?

To answer these research questions, we conduct an experimental study on *few-sample* named entity recognition for vulnerability reports, which, to our knowledge, has not been explored in prior work. We focus on the `VIEM` dataset (Dong et al., 2019), which contains vulnerability reports with 3 types of entities (software name, software version, and outside) across 13 sub-datasets organized by vulnerability category. For the largest sub-dataset, we find that through fine-tuning pre-trained language models, it is possible to match state-of-the-art performance (0.92 to 0.93 F1 score (Dong et al., 2019)) using only 576 labeled sentences for training, or 10% of the original training data. For the other 12 categories, the same approach matches state-of-the-art performance with only 11.2% of labeled samples. Our experiments also show that simple fine-tuning outperforms state-of-the-art few-shot NER systems (Yang and Katiyar, 2020).

The main contributions of this chapter are:

- We propose, for the first time, to study the problem of named entity recognition (NER) for security vulnerability reports in a few-sample setting;
- We perform an experimental study by fine-tuning three state-of-the-art language models and evaluating one metric learning-based method for few-sample NER in vulnerability reports, showing that fine-tuning can achieve similar performance using only 10% or 11.2% of the original training dataset;
- We discuss multiple future research opportunities for information extraction from security vulnerability reports.

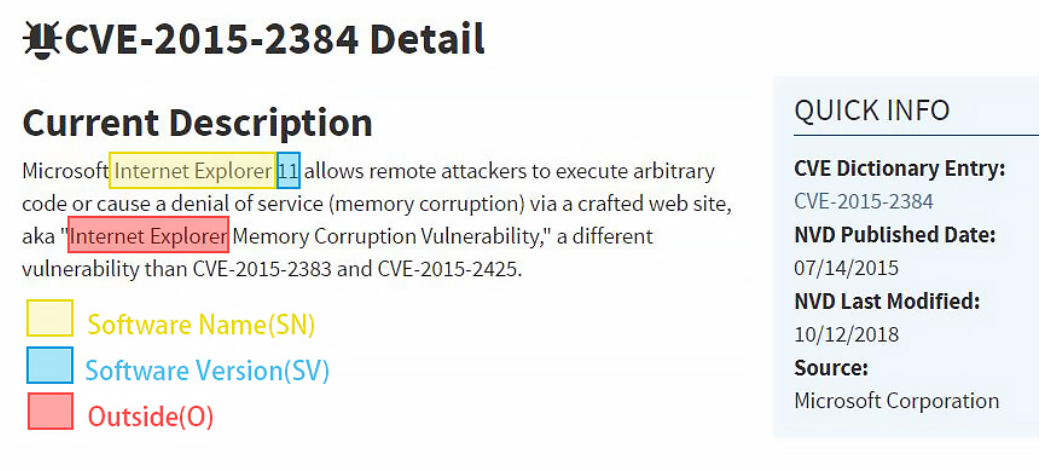


Figure 3.1: An example of a public security vulnerability report (CVE-2015-2384)

3.2 Problem Definition and Challenges

3.2.1 Few-Sample Named Entity Recognition

Named entity recognition (NER) is the task of assigning predefined entity names to tokens in input text. Formally, with predefined tagging set $\mathcal{Y}$ and a sequence of tokens (input sentence) $X_i = [x_{i1}, x_{i2}, \cdots, x_{iT}]$, each x_{ik} $(k = 1, 2, \cdots, T)$ corresponds to a tag $y_{ik} \in \mathcal{Y}$ specifying its entity type. This gives X_i a tagging sequence $Y_i = [y_{i1}, y_{i2}, \cdots, y_{iT}]$ and a typical NER dataset is of the form $\mathcal{D} = \{(X_i, Y_i)\}_{i=1}^{N}$.

There are two distinct definitions of what few-shot learning algorithms aim to achieve. In the first definition, the goal is to match performance when only a subset of the training data is available; in the second, the test set contains unseen classes not in the training set, and the algorithm needs to generalize to them. In this work, we focus on the first setting.

3.2.2 Named Entity Recognition for Vulnerability Reports

An example of a security vulnerability report (CVE-2015-2384) is shown in Figure 3.1, which is a snapshot from the NVD website. In this work, we focus on the `VIEM`

dataset (Dong et al., 2019), which contains 13 vulnerability categories and two types of named entities: software names and software versions. The tag set $\mathcal{Y}$ is instantiated as $\{\texttt{SN}, \texttt{SV}, \texttt{O}\}$, where `SN`, `SV`, `O` refer to software name, software version, and non-entities, respectively. For example, the CVE-2015-2384 report warrants a security alert, as the `Internet Explorer` version `11` software might expose a vulnerability to attackers. The training, validation, and testing sets share the same tag set $\mathcal{Y}$.

3.2.3 Data-Specific Challenges

There exist several data-specific challenges in the `VIEM` dataset.

Contextual Dependency

When identifying vulnerable software names and versions in vulnerability reports, the tagger must consider the entire context. For example, in CVE-2015-2384, `Internet Explorer` is tagged as a software name for its first occurrence; in the second occurrence, it appears in the vulnerability name *Internet Explorer Memory Corruption Vulnerabilities* and is therefore no longer a software name. Software version tags are also context dependent: in different CVE entries, the same version number may be tagged `SV` or `O` depending on whether the accompanying software name is tagged as an entity.

Data Imbalance

The goal of an NER system is to identify entities of interest and tag all other tokens as outsiders. It is therefore possible that entire sentences include only outsider tokens. In general-domain NER datasets such as CoNLL 2003 and OntoNotes 5, the proportion of such sentences in the training sets is relatively low (around 20-50%). In vulnerability reports, however, the proportion of all-`O` sentences is high (over 75% in

some categories), as shown in Table 3.1. Such data imbalance can cause the classifier to trade off minority-group performance for majority-group performance, leading to higher error rates for minority samples.

Table 3.1: The proportion of sentences in the training set that include only non-entities

Dataset	Domain	Non-entity-only sentences
CoNLL 2003	General	20.71%
OntoNotes 5	General	51.60%
`VIEM` (all)	Security	60.34%
`VIEM` (w/o `memc`)	Security	75.44%

3.3 Methods

In this section, we introduce the methods investigated in this chapter. The first method leverages fine-tuning of pre-trained language models (PLMs); the second is a few-shot NER method based on nearest-neighbor search.

3.3.1 Fine-Tuning Pre-trained Language Models

Selection of Pre-trained Language Models

We investigate three models from the HuggingFace model hub: `bert-large-cased`, `roberta-large`, and `electra-base-discriminator`. BERT (Devlin et al., 2019) was the first pre-trained language model to achieve strong performance relative to prior art; RoBERTa (Liu et al., 2019) and Electra (Clark et al., 2020) are among the top-ranked language models on the GLUE leaderboard. For BERT, we choose the cased version `bert-large-cased` rather than the uncased counterpart, as token casing could be informative for identifying software names (`SN`). Our implementation is based

on the HuggingFace `transformers` library (Wolf et al., 2019) (4.5.1), leveraging the `AutoModelForTokenClassification` class for fine-tuning.

Fine-Tuning on the `memc` Category

Following the experimental protocol of the `VIEM` system, the PLM is first fine-tuned on the vulnerability category of `memc` (short for "memory corruption") as this category suffers less from data imbalance; fine-tuning on this category could therefore best help PLMs learn representations specific to the computer security domain. However, rather than allowing PLMs access to the entire `memc` training dataset, we constrain access to 1%-10% of the dataset.

Transfer Learning to Other Categories

Previous works on fine-tuning language models show that fine-tuning can benefit from multi-task learning or transfer learning, i.e., fine-tuning on a related task before fine-tuning on the objective task (Phang et al., 2018). We adopt this approach: the PLM is first fine-tuned on the `memc` category, then we start from that checkpoint and continue fine-tuning on the other 12 categories. We again limit the number of annotated samples during transfer learning. Throughout the transfer learning experiment, we sample varying numbers of training samples from each of the 12 categories and combine the subsets to form an *aggregate* training set. For validation and testing, we evaluate the model on each category *independently*.

Hyperparameter Optimization

The hyperparameters for fine-tuning include learning rate, batch size, and number of training epochs. Improperly setting these hyperparameters may degrade performance, making hyperparameter tuning critical. We leverage grid search where the learning

rate is selected from $\{1e\text{-}6, 5e\text{-}6, 1e\text{-}5\}$; the number of training epochs is chosen from $\{3, 5\}$; batch size is fixed to 2 per device.

Setting the Random Seed

Fine-tuning performance for PLMs is known to be unstable, especially under few-sample settings. Some seemingly benign parameters, such as the choice of random seed, could introduce substantial variance in model performance. The choice of seed could influence the initialization of weights for task-specific layers when applying PLMs to downstream tasks. When these task-specific layers are initialized poorly, and the training set is small, the network is unlikely to converge to a good solution without extensive training. We set the seed for model training to 42 and randomly restart each experiment 10 times, reporting the average scores.

3.3.2 Few-shot Named Entity Recognition

After trying fine-tuning methods, we found they still require hundreds of training examples. One question is whether we can further reduce the number of training labels to a few. The `StructShot` system (Yang and Katiyar, 2020) manages to beat competitive systems with only 1 to 5 annotated named entities per entity class.

The key technique used in `StructShot` is nearest neighbor search. Rather than directly training a PLM with a token classification objective, their system is first fine-tuned on a high-resource, yet distantly related dataset, and then the model is used as a feature extractor to obtain token representations for both test tokens and a small support set. Test token tags are determined by retrieving the tag of the most similar token from the support set. A CRF decoding procedure uses similarities estimated in previous steps to produce the optimal tag sequence.

3.4 Experiments

3.4.1 Datasets

We experiment on the `VIEM` dataset (Dong et al., 2019). This dataset is collected from public vulnerability databases (CVE) and numerous security forums. The goal is to identify vulnerable software names and software versions. The tag set $\mathcal{Y}$ includes `SN` (software names) and `SV` (software versions), with the `O` (outside) tag for all other tokens. The dataset is manually labeled by three security experts to minimize labeling errors and enforce consistent annotation standards. Records span 20 years and contain all 13 categories listed on the CVE website.

Table 3.2: Statistics of the `VIEM` dataset

Category	`memc`			Avg. of other 12		
Split	train	valid	test	train	valid	test
Number of samples	5758	1159	1001	468	117	556
Sentence-level entity proportion	0.56	0.33	0.46	0.24	0.27	0.26
Token-level entity prop. (`SN`)	0.061	0.037	0.056	0.021	0.024	0.024
Token-level entity prop. (`SV`)	0.082	0.081	0.079	0.031	0.033	0.033

From the statistics shown in Table 3.2, we observe dataset imbalance at different levels:

- **Sample Number Imbalance**: The `memc` category has significantly more samples in every split than other categories.
- **Entity Number Imbalance**: The `memc` category has predominantly higher entity proportion in both sentence and token levels.

- **Entity Type Imbalance**: The appearances of `SV` tokens are generally more frequent than `SN`.

Preparing the Few-Sample Dataset

Among the 13 vulnerability categories, only `memc` includes an official validation split. To enable hyperparameter optimization across other categories, we randomly sample 10% of the training samples as a held-out validation set.

We create training and validation sets by sampling subsets of the full dataset:

- **Fine-Tuning**: We vary the proportion of sampled data from the `memc` training set from 1% to 10% (equivalent to sample sizes from 58 to 576).
- **Transfer Learning**: We vary the number of samples in other categories from $\{32, 64, 128, 256\}$ per category.

For validation sets, we ensure they have the same size as training sets whenever possible, since validation samples also require annotations and consume labeling budget. Throughout data preparation, we fix the random seed to 42.

3.4.2 Evaluation Metrics

Following the evaluation metrics used in prior work, we use precision, recall, and F1 score to evaluate our NER system.

3.4.3 Experimental Setup

Evaluated Methods

Our experiments contain two parts: first, we evaluate few-sample learning performance on the `memc` category; second, we evaluate performance on the other 12 categories.

For the `memc` category, we fine-tune models with varying proportions from 1% to 10%. We investigate whether fine-tuning with a small proportion of samples is feasible for PLMs to reach competitive performance.

For other categories, we compare four settings:

- **FT**: Direct application of models fine-tuned on `memc` to other categories.
- **FT+SS**: After PLM is fine-tuned on `memc`, it is coupled with `StructShot` and directly applied to other categories without transfer learning.
- **FT+TL**: Apply transfer learning on other categories using the best fine-tuned checkpoint from `memc`.
- **FT+TL+SS**: Built upon **FT+TL** by applying `StructShot` after both fine-tuning and transfer learning.

3.4.4 Experimental Results: Fine-Tuning on `memc`

Summary of Main Results

Fine-tuning performance is detailed in Table 3.3. We observe that the F1 difference from the `VIEM` system is within about 1.5% for `SN` and within 1% for `SV`. Importantly, the amount of data we use is only 10% of the `VIEM` system.

Table 3.3: Fine-tuning results for the `memc` category

	SN prec.	SN rec.	SN F1	SV prec.	SV rec.	SV F1
`VIEM` (full data)	0.977	0.992	0.984	0.988	0.992	0.990
BERT (10% data)	0.958	0.981	0.969	0.982	0.985	0.983
RoBERTa (10% data)	0.957	0.981	0.969	0.984	0.984	0.984
Electra (10% data)	0.968	0.975	0.972	0.983	0.989	0.986

To investigate how fine-tuning helps PLMs adapt to the security domain and pick up task-awareness, Figure 3.2 shows a 2D projection of token-level hidden representations obtained from Electra for the `memc` training set before and after fine-tuning. We can see much more compact clusterings of `SN` and `SV` after fine-tuning; fine-tuning informs PLMs of underlying patterns in task data.

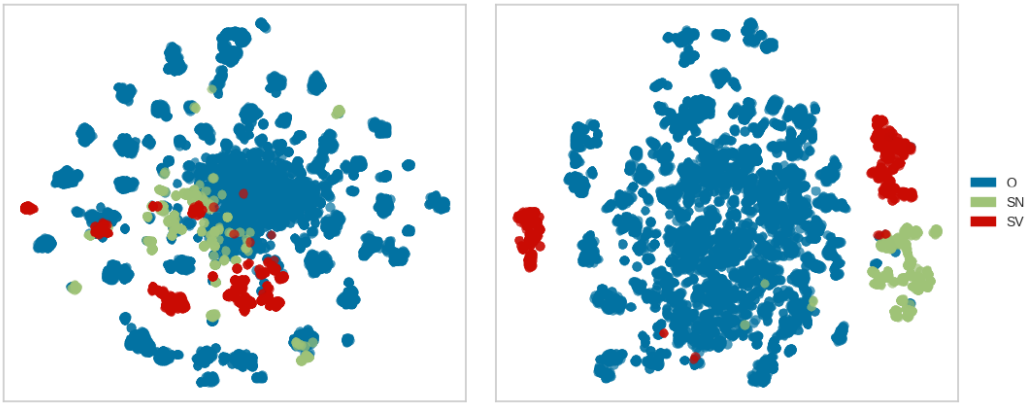


Figure 3.2: The t-SNE visualization of the `memc` training set at the token level before (left) and after (right) fine-tuning. Fine-tuning produces more compact clusters for both `SN` and `SV`.

Fine-Tuning with Different Training Sample Sizes

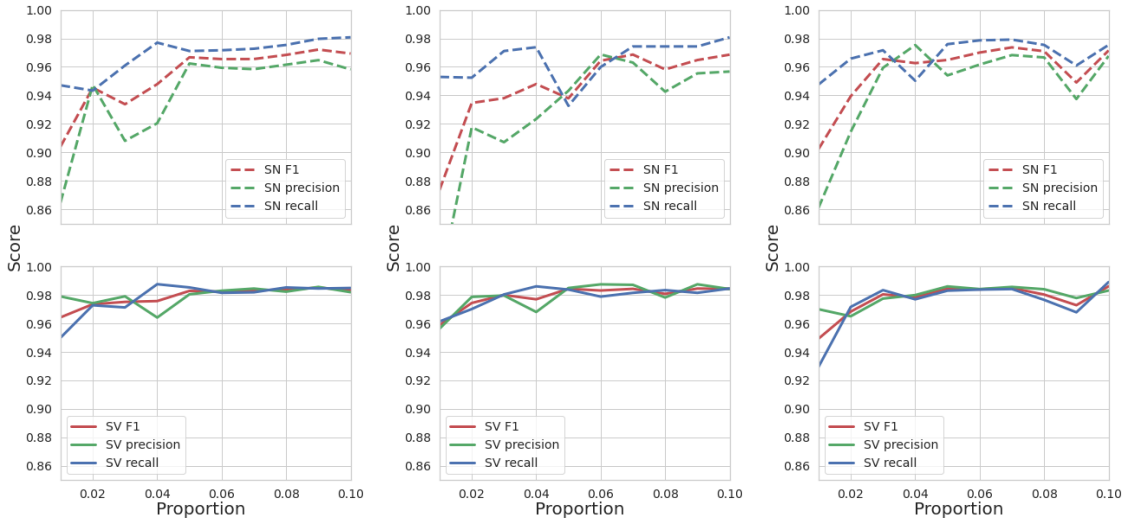
We vary the number of labeled samples per category from 1% to 10% to see how training set size affects fine-tuning performance.

As shown in Figure 3.3, when the proportion of labeled training samples increases from 1% to 10%, performance generally improves, but not monotonically. For example, fine-tuning RoBERTa with 5% of data sometimes performs worse than with only 3%.

There is also disparity in fine-tuning performance between `SN` and `SV`. Specifically, `SV` can achieve around 95% F1 with as little as 1% of training samples available. This does not hold for `SN`: with under 1% training data availability, F1 for `SN` does

not exceed 92% for BERT; for RoBERTa and Electra, this metric is below 88%. The disparity suggests that correctly tagging `SV` is easier than tagging `SN`, possibly because software version naming conventions exist that taggers can exploit, whereas similar conventions do not exist for software names.

Another general tendency is the relation between precision and recall: across different models and tags, recall is generally higher than precision. This is desirable for security applications as incorrectly tagging regular software as vulnerable only leads to more manual verification effort, while missing vulnerabilities could leave critical systems unattended.



(a) BERT (b) RoBERTa (c) Electra

Figure 3.3: The influence of training set size on fine-tuning performance

3.4.5 Experimental Results: Transfer Learning on Other Categories

Summary of Main Results

As shown in Table 3.4, transfer learning with aggregation of 64 samples per category matches `VIEM` system performance; the absolute difference is less than 1% in F1 score.

Despite less than 1% improvement over our system, the `VIEM` system is trained on the entire dataset of 7016 samples. In terms of data utilization, we reduce the number of annotations by 88.80%.

`StructShot` can help with *precision* of `SN` but struggles with *recall*. Introducing `StructShot` has led to a degradation in the F1 score, casting doubt on its applicability to our task. Specifically, the absolute gap in F1 score can be up to 20%. This gap still exists even with additional transfer learning (**FT+TL+SS**).

Table 3.4: Transfer learning results averaged across 12 categories

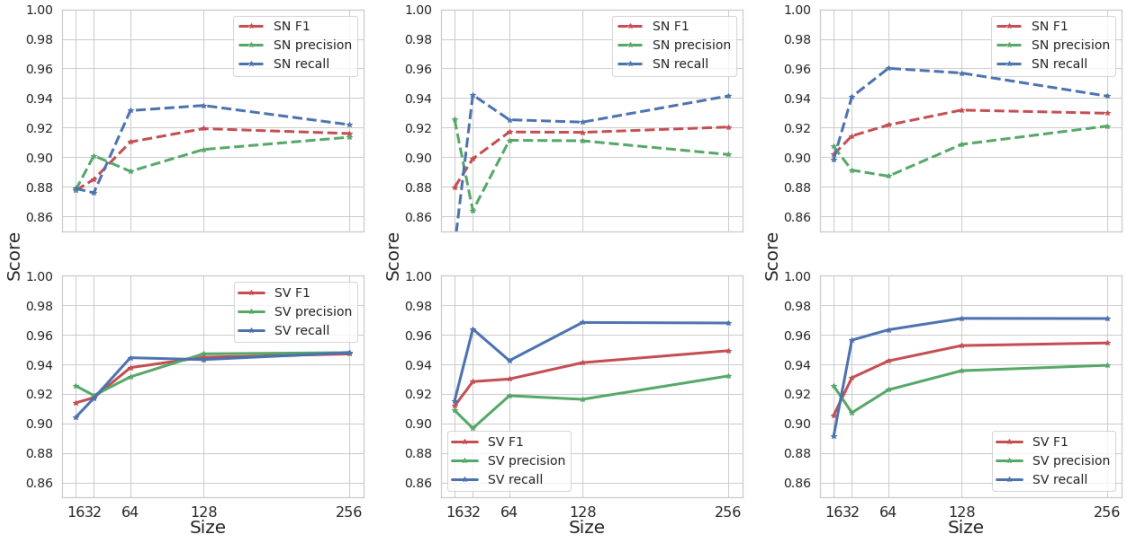
	SN prec.	SN rec.	SN F1	SV prec.	SV rec.	SV F1
`VIEM` (w/o transfer)	0.828	0.849	0.838	0.843	0.941	0.889
`VIEM` (w/ transfer)	0.918	0.929	0.924	0.938	0.941	0.940
BERT						
FT	0.876	0.795	0.832	0.881	0.934	0.906
FT+SS	0.904	0.684	0.762	0.900	0.788	0.818
FT+TL	0.895	0.930	0.912	0.937	0.935	0.936
FT+TL+SS	0.906	0.764	0.777	0.937	0.914	0.924
RoBERTa						
FT	0.856	0.851	0.852	0.882	0.931	0.905
FT+SS	0.891	0.681	0.753	0.911	0.849	0.874
FT+TL	0.875	0.941	0.906	0.916	0.956	0.935
FT+TL+SS	0.884	0.785	0.799	0.919	0.936	0.927
Electra						
FT	0.866	0.824	0.842	0.885	0.925	0.904
FT+SS	0.869	0.681	0.727	0.836	0.743	0.742
FT+TL	0.900	0.944	0.921	0.926	0.949	0.937
FT+TL+SS	0.893	0.826	0.832	0.904	0.948	0.923

Transfer Learning with Different Training Sample Sizes

We conduct a similar performance analysis when varying the training sample size, as in fine-tuning; we visualize average performance metrics because we apply transfer

learning to fine-tuned models across 12 categories. As shown in Figure 3.4, when sample numbers range from 32 to 256, more samples contribute to transfer learning improvements, but gains are less clear beyond 64 per category: performance reaches a plateau with more than 64 samples.

We observe similar performance disparity between `SN` and `SV` as in fine-tuning. Specifically, we can use as few as 16 samples per category to achieve around 91% F1 for `SV`, while similar performance is not attainable for `SN` until we enlarge the training set to 64 (for BERT and RoBERTa) or 128 (for Electra) samples per category.



(a) BERT (b) RoBERTa (c) Electra

Figure 3.4: The influence of training set size on transfer learning performance

Analysis on Data-Specific Challenges

We discussed that the `VIEM` dataset contains challenges in *context dependency* and *dataset imbalance*. Even though we do not explicitly address either, our experiments show they have not largely affected performance:

- **Dataset Imbalance**: The average proportion of sentences with entities in the

training set is only 24.35%; this might concern performance due to sparsity of informative tokens. However, performance matches or exceeds that of the `VIEM` system, with average F1 scores above 0.9, indicating that dataset imbalance does not degrade effectiveness.

- **Context Dependency**: Our experiments show that transfer learning can help address the context dependency problem. For example, models without fine-tuning misclassify certain software names as non-entities, but after transfer learning correctly identify them in their proper context.

Troubleshooting StructShot

The unsatisfactory performance of `StructShot` shown in Table 3.4 warrants caution about scenarios where it can be applied. We conduct error analysis to pinpoint its weakness. In controlled experiments, we find that `StructShot` is sensitive to adversarial examples (Figure 3.5).

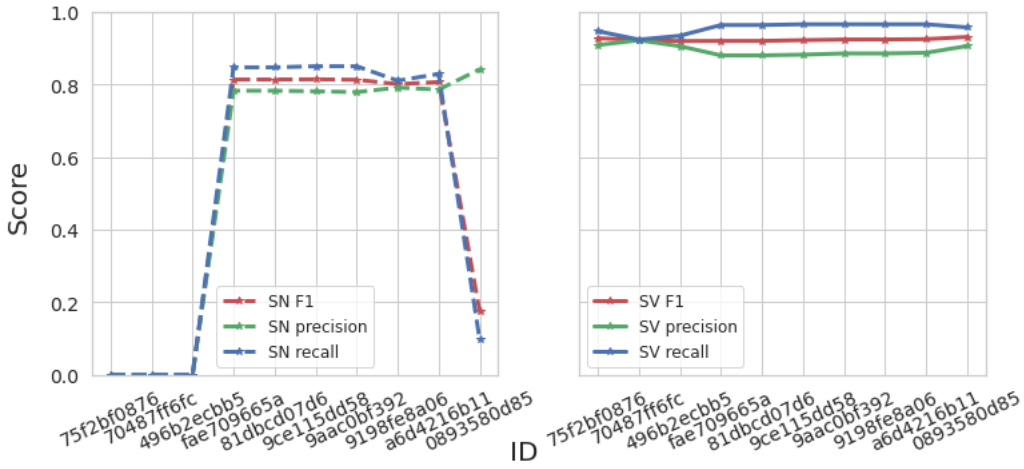


Figure 3.5: `StructShot` is sensitive to adversarial samples. When testing on the `csrf` category, an additional sample in the support set leads to 70% F1 score degradation for `SN`.

When we manually create the support set by sequentially adding samples, a

single adversarial sample can reduce overall performance by more than 70%. One benefit of `StructShot` reported in prior work is its improved ability to assign non-entity tags, since these tokens do not carry unified semantic meaning. However, because of *context dependency* in NER for public security reports, non-entity tokens in one context might be entities in another, causing confusion for the nearest-neighbor-based tagger.

3.5 Related Work

3.5.1 Information Extraction in Public Vulnerability Databases

The `VIEM` system (Dong et al., 2019) is among the first works to extract information from public vulnerability databases using a combination of named entity recognition (NER) and relation extraction (RE). The extracted information is used to identify vulnerabilities and inconsistencies between major security databases.

Prior work on information extraction in the security domain has mostly focused on other aspects, such as dataset curation, data analysis, and visualization. These include creating vulnerability datasets from CVE data, estimating severity from CVE descriptions, and building interactive visualization tools to explore vulnerabilities.

Our work builds on the `VIEM` system's setting and improves vulnerability identification by minimizing data labeling costs while matching the performance of existing systems.

3.5.2 Named Entity Recognition for Computer Security

NER has extensive applications in computer security tasks. Researchers have emphasized the importance of NER applications in software domains for developing security applications, such as detecting compromised and vulnerable devices. Other works in-

vestigate challenges of NER involving software, noting that common word polysemy complicates the task. NER has been used to identify software names in mobile application descriptions to locate security permissions, to automatically extract relevant information from ExploitDB descriptions to create CVE records, and to extract APIs from software documentation.

Our work strives to identify vulnerabilities from public security reports and therefore treats NER as a vehicle. However, we face a unique challenge: resource constraints; our system should achieve competitive performance with as few labeled samples as possible.

3.5.3 Few-Sample Named Entity Recognition

The study of named entity recognition dates back to Message Understanding Conference-6 (MUC-6) held in 1995 (Grishman and Sundheim, 1996), where researchers first identified the problem of recognizing named entities using rule-based or probabilistic approaches. The adoption of neural approaches and the notion of few-sample NER only happened in recent years.

The success of neural sequence labeling created tension between the cost of obtaining annotations and the data-hungry nature of neural model training. The notion of few-shot learning, originally proposed in the computer vision community (Finn et al., 2017), has also been adopted by NLP practitioners. Methods based on prototypical networks (Snell et al., 2017) and nearest neighbor search (Yang and Katiyar, 2020) have been applied to NER. Recent work shows the potential of pre-trained language models to reduce annotation effort by leveraging universal representations of language (Devlin et al., 2019; Liu et al., 2019).

Our work is consistent with prior efforts aimed at reducing the number of training labels required. However, unlike prior approaches, we discard prototype-based tag

assignment schemes and instead propose directly applying PLMs via fine-tuning and transfer learning. Empirical evidence shows that this design choice achieves better, more stable performance metrics.

3.6 Conclusions and Future Work

In this chapter, we take a first attempt to investigate the problem of few-sample named entity recognition (NER) for public security vulnerability reports. By leveraging fine-tuning of three pre-trained language models (BERT, RoBERTa, Electra), we find it is possible to match prior art performance with much less labeled training data: first, for vulnerability reports in the `memc` category, fine-tuning on PLM matches state-of-the-art scores using only 10% of training examples (576 labeled sentences); second, for vulnerability reports in other categories, through fine-tuning and transfer learning, it is possible to match prior scores using only 11.2% of training examples (64 labels per category). Few-sample learning has effectively reduced the training data required for NER for security vulnerability reports.

It is important to note that this work is just a beginning, and great potential exists for further improvement. Future work includes:

- **Leveraging Unlabeled Data**: This work only considers labeled samples. It is worth exploring how to leverage unlabeled data in the security domain to help models more quickly adapt to in-domain tasks.
- **Leveraging External Knowledge Bases**: Our experiments show it is more difficult to achieve good F1 for software names than software versions, because there is a higher chance for software names in testing data to be completely unseen. Leveraging external knowledge bases could help recognize unseen software names.

- **Adversarial Learning**: We observed adversarial cases that cause dramatic performance drops in few-sample transfer learning. Investigating adversarial learning could improve robustness.

Chapter 4

Generating Behavioral Tests for NLP Models

The previous chapter used a fine-tuned model to extract structured facts from text. This chapter takes the next step on both axes (Figure 1.1): on the software development life cycle, it moves to *software testing*, and on the axis of autonomy, it takes the second step, GENERATE, where the model no longer only reads text but writes it, producing test cases on demand. The technique is in-context generation with a large language model (Chapter 2). We present TestAug, which generates capability-based test cases for language classifiers, and HateModerate, which grounds the same idea in a real content-moderation policy. The bottleneck it removes is manual test suite authoring.

4.1 Introduction

In recent years, NLP has seen major breakthroughs in model performance. Conventional approaches to evaluating NLP models rely on reporting aggregate metrics such as accuracy and F1 scores on held-out datasets. However, held-out scores do not represent model performance on real-world data (Geva et al., 2019; Gururangan et al., 2018; Bai et al., 2021). Moreover, aggregated scores cannot show where models fail and how to fix the failures. Recent studies show that even stress-tested commercial NLP APIs often fail on simple behavioral test cases (Glockner et al., 2018; Ribeiro et al., 2020).

To assist developers in finding behavioral failures in their models, recent work has proposed a framework called CheckList for *behavioral* or *capability-based NLP testing* (Ribeiro et al., 2020; Tarunesh et al., 2021). Such tests include input-output

pairs to assess model performance for each *linguistic capability*. For example, a test case containing a negated positive word tests whether a sentiment classifier correctly understands negation. With a set of sentences, each containing negated positive words, we can test whether the model correctly handles such sentences. Such a set is called a *test suite*.

In existing capability-based testing frameworks (Ribeiro et al., 2020; Tarunesh et al., 2021), test suites are generated from manually created natural-language templates and predefined word lists to fill them. Existing work thus has two disadvantages:

- **High Cost of Labeling**: Creating templates requires high cost and expert annotations that understand both linguistic rules and potential NLP pitfalls.
- **Low Diversity**: Test cases generated through templates often show diversity only on a superficial level, such as varying only person names.

In this chapter, we propose a novel framework for capability-based testing to address these challenges. In our framework, TestAug, the developer first annotates a few seed test cases, then TestAug leverages the GPT-3 engine (Brown et al., 2020) to generate similar test cases. Next, TestAug expands the GPT-3-generated cases into templates, thereby generating more cases. Finally, TestAug includes a validity classifier to check the correctness of generated cases and discard invalid ones. Our experiments show that validity classifiers filter invalid cases with success rates of at least 80-90%. Furthermore, valid generated cases are more diverse and can detect more bugs than existing work (Ribeiro et al., 2020).

We then extend this approach to content moderation with HateModerate, a framework that tests hate speech classifiers against Facebook's 41 community con-

tent policies (Facebook, 2022)[1]. This application demonstrates how capability-based testing can be grounded in real-world platform requirements, revealing substantial gaps between model behavior and policy guidelines.

4.2 Problem Definition

Software testing involves identifying inconsistencies between software's actual and expected behavior (Zhang et al., 2019). *Behavioral testing* examines external software behaviors and requires developers to collect test cases (input/output pairs) to constitute a *test suite* for testing specific behaviors.

Following the success of NLP, behavioral testing was introduced to test NLP models (Ribeiro et al., 2020), especially large language models. The expected behaviors are defined in several aspects called *capabilities*. For a sentiment classification model, we expect it to output negative sentiment for input containing negated positive words, e.g., "I don't like the food." Behavioral testing goes beyond held-out validation, allowing developers to detect and monitor behavioral failures that aggregate performance metrics may miss.

In CheckList (Ribeiro et al., 2020), around 10 capabilities are defined for each NLP task. For each capability, several tests are created, with each test described in natural language. Each test contains one or more templates with slots and predefined word lists. For example, for negation capability, one template is "[it] [benot] [a:pos_adj] [air_noun]." By defining a list for each slot, developers generate test cases such as "That is not a perfect seat."

[1]Following Meta's rebranding, the URL for these community standards has changed. The current guidelines can be found at https://transparency.meta.com/policies/community-standards/hateful-conduct/.

4.3 The TestAug Framework

Given a linguistic capability and specific tests, previous approaches rely on manual templates. We propose TestAug (Test Suite Augmentation) to reduce manual effort. Figure 4.1 shows our framework's control flow.

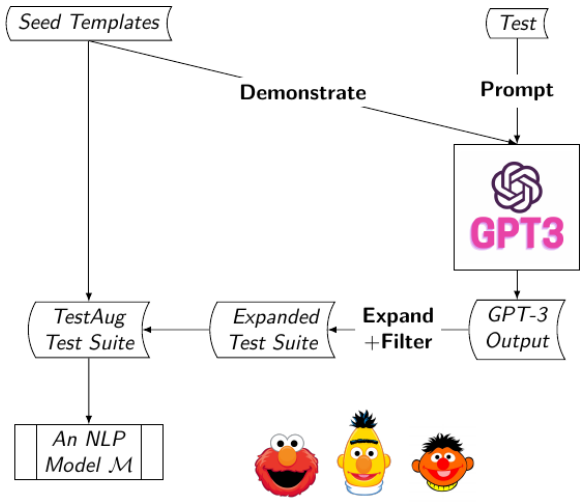


Figure 4.1: The control-flow graph of TestAug

First, TestAug starts with a few seed test cases from CheckList (Ribeiro et al., 2020). It leverages the test description and seed cases to prompt GPT-3 to generate more cases. Generated cases are examined by a trained binary classifier and expanded into additional templates by matching them to seed cases. Finally, the aggregate test suite is used to test the model.

4.3.1 Prompt Engineering for GPT-3

We design prompts based on practices for instructing GPT-3 in dataset creation (Liu et al., 2022; Reif et al., 2022; West et al., 2022; Schick and Schütze, 2021; Reynolds and McDonell, 2021). A prompt is an instruction that tells GPT-3 what to generate. For example, "A negative sentiment sentence with negated positive word." Generative

models' performance improves with *demonstrations* (Gao et al., 2021), i.e., example sentences that show what generated sentences should look like.

We propose to use the test description as the prompt, followed by three randomly sampled seed test cases as demonstrations. We format each demonstrated case with brackets "{}", as illustrated in Figure 4.2. This format encourages GPT-3 to generate valid sentences with higher probabilities. Without brackets, GPT-3 tends to generate long dissimilar sentences; for two-sentence tasks, GPT-3 is less likely to generate correctly formatted pairs.

```
A negative sentiment sentence with negated positive word.
- { No one enjoys that pilot. }
- { No one admires the seat. }
- { No one appreciates that airline. }
- { No one appreciates that air traffic controller. }
```

Figure 4.2: Prompt design for sentiment classification test case generation. The test description specifies context; seed sentences help GPT-3 generate similar yet diverse cases.

4.3.2 Filtering Incorrect Test Cases

Test cases generated by GPT-3 may fail to satisfy tests: they may not satisfy the required format (e.g., paraphrase detection requires sentence pairs while GPT-3 generates only one), or they may not satisfy test requirements (e.g., a generated sentiment test case does not actually have negative sentiment).

To address these issues, we create a validity filter that automatically removes invalid test cases. The filter is a binary classifier trained on manually annotated GPT-3-generated sentences. We follow a two-phase process: in the first phase, we generate 30-50 cases per test description and manually annotate validity. Tests with predominantly valid cases ($\geq$90% for sentiment and paraphrase, $\geq$80% for NLI) pass

through; otherwise, we proceed to phase two. In the second phase, we generate more sentences until we have at least 100 invalid and 100 valid cases for training a RoBERTa classifier. After training, we can use classifiers with GPT-3 in a fully automatic manner.

4.3.3 Expanding GPT-3 Cases into Templates

After obtaining generated cases, we can further augment the test suite by expanding GPT-3 cases into templates. If a word in a seed case reappears in a GPT-3 case, it can be converted back to a slot, and we vary slot words using pre-defined lists. For example, using template "No one [pos_verb_present]s [the] [air_noun]." and word "appreciates" for [pos_verb], we create seed "No one appreciates that airline." GPT-3 generates "No one appreciates that air traffic controller." Since "appreciates" reappears, we create a new template "No one [pos_verb] that air traffic controller." We only consider content words (nouns, verbs, adjectives) to avoid creating nonsensical sentences from misplaced pronouns.

4.4 Experiments

We evaluate TestAug's effectiveness and compare it with existing work in multiple aspects: ability to detect model failures, diversity of test cases, validity of generated cases, and manual effort saved.

4.4.1 Experimental Settings

Evaluated Tasks. We compare with existing work on sentiment classification (SST dataset (Socher et al., 2013)), paraphrase detection (QQP dataset), and natural language inference (SNLI dataset (Bowman et al., 2015)). We skip extractive QA because

GPT-3 struggles to generate cases in which all components (context, question, answer) meet the requirements simultaneously. We skip hate speech detection because GPT-3 cannot be prompted to generate profanity words.

Evaluated Models. We test only open-source models (BERT, RoBERTa, ALBERT, DistilBERT) because commercial APIs are inaccessible for fine-tuning, which is critical for our evaluation. We use publicly available fine-tuned models from HuggingFace.

4.4.2 Evaluating Bug Detection Ability

Metric: Patched Failure Rate. To compare test suite effectiveness, we propose the following methodology: merge test suites into a large suite, randomly partition it into training and testing sets, fine-tune/patch the model using training data from each suite, and compare failure rates on the common test set. A lower patched failure rate indicates stronger bug detection ability.

Results. Table 4.1 shows average failure rates across all capabilities. We observe that TestAug's failure rate is consistently lower than that of CheckList, indicating that TestAug finds more bugs. GPT-3-generated cases are more important than expanded cases, whereas expanded cases are not always helpful.

Per-Capability Breakdown. Figure 4.3 reports the per-capability error rates for paraphrase detection and natural language inference, complementing the aggregate numbers in Table 4.1. The breakdown shows that the gains over CheckList are spread across most individual capabilities rather than driven by a few.

4.4.3 Evaluating Diversity

We use two metrics to evaluate diversity: Self-BLEU (Zhu et al., 2018) (lower values indicate higher diversity) and the number of unique dependency paths (Jurafsky and

Table 4.1: Comparison of bug detection ability between TestAug and CheckList. Each cell shows the failure rate (error rate on the held-out validation set).

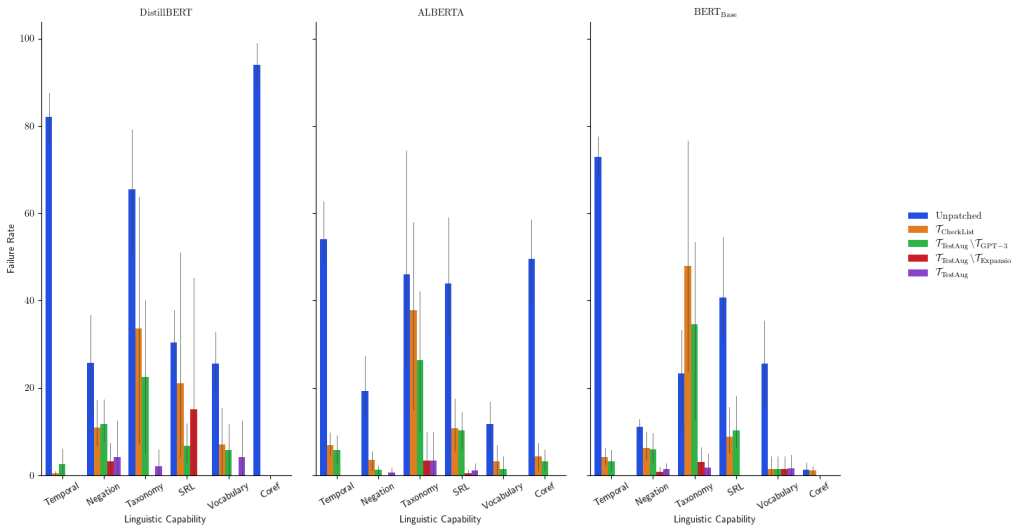
Model	Original	Unpatched	CheckList	TestAug
Sentiment Classification				
ALBERT	7.3	32.6	13.4	**11.3**
$\text{BERT}_{\text{Base}}$	7.6	33.9	9.0	**8.3**
DistilBERT	10.0	29.5	6.5	**3.9**
$\text{RoBERTa}_{\text{Base}}$	5.7	14.2	3.7	**1.6**
Paraphrase Detection				
ALBERT	9.3	38.1	7.1	**0.6**
$\text{BERT}_{\text{Base}}$	9.1	36.0	6.2	**0.5**
DistilBERT	10.3	49.8	12.5	**1.1**
Natural Language Inference				
ALBERT	9.9	42.8	30.1	**23.0**
DistilBERT	12.6	34.7	23.6	**16.5**
$\text{RoBERTa}_{\text{Large}}$	8.1	17.8	8.3	**8.0**

Martin, 2000) (higher values indicate higher diversity).

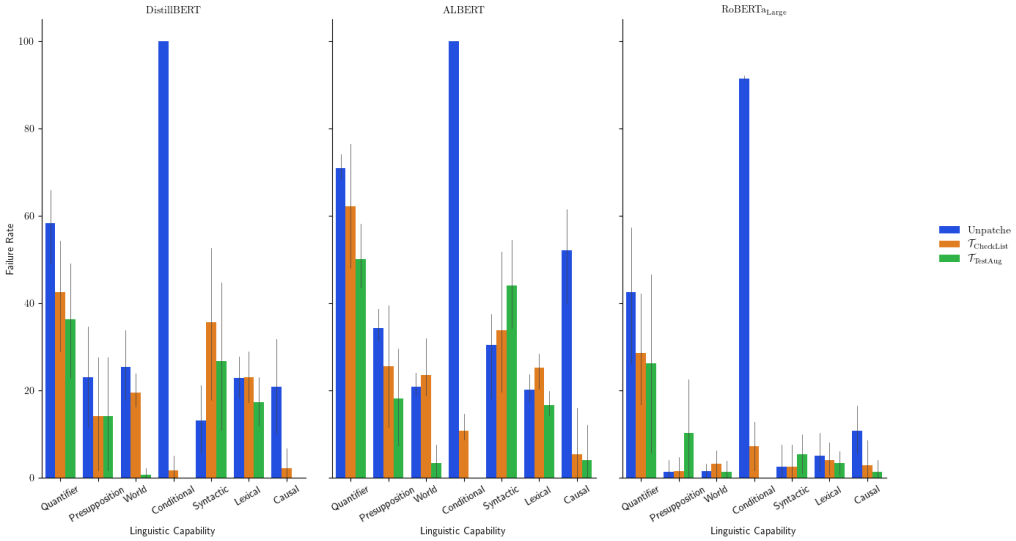
Table 4.2: Linguistic diversity of test suites

	Self-BLEU4 (↓)	Unique Dep. Paths (↑)
Sentiment Analysis		
TestAug	0.634	480
CheckList	0.853	84
Paraphrase Detection		
TestAug	0.627	418
CheckList	0.775	117
NLI		
TestAug	0.430	210
CheckList	0.454	116

From Table 4.2, we observe substantial improvement in linguistic diversity: Self-BLEU4 decreases by at least 5.3% and unique dependency paths increase by at least 1.81 times.



(a) Paraphrase Detection



(b) Natural Language Inference

Figure 4.3: Capability-wise error rates on paraphrase detection and natural language inference tasks.

4.4.4 Evaluating Validity and Manual Effort Saved

The two-phase approach successfully filters out invalid cases, with classification accuracy consistently exceeding the validity thresholds. Cohen's κ on the test set annotation indicates substantial agreement (Cohen's κ is a chance-corrected measure

of how often two annotators agree, where values above 0.6 are conventionally called substantial) (McHugh, 2012).

As shown in Table 4.3, when querying GPT-3 with a few hundred sentences per task, we obtained new valid test cases 11.2 to 50.4 times in size. Template expansion increased available templates by at least 49.7 times, reducing manual effort by at least 98.0%.

Table 4.3: TestAug saves manual efforts in generating new test cases

Seed Sentences	New Sentences	Seed Templates	New Templates
Sentiment Classification			
292	3275 (11.2×)	29	1441 (49.7×)
Paraphrase Detection			
124	6427 (50.4×)	17	1307 (76.9×)
Natural Language Inference			
150	4976 (33.2×)	50	/

4.5 Application: Testing Hate Speech Detectors Against Content Policies

We extend the capability-based testing approach to content moderation. Existing hate speech datasets typically provide binary labels without grounding examples in specific policy violations. This makes it difficult to assess whether trained models would correctly enforce a platform's actual moderation policies.

4.5.1 The HateModerate Framework

We propose HateModerate, a dataset for testing hate speech detectors against content policies. We focus on Facebook's 41 community standards guidelines for hate speech (Facebook, 2022), which represent the most comprehensive platform policies (Jiang et al., 2020). These guidelines are organized into four tiers based on content severity: Tier 1 includes the most offensive content (e.g., dehumanization and violence), while Tiers 2-4 are progressively less severe (e.g., stereotyping and contempt).

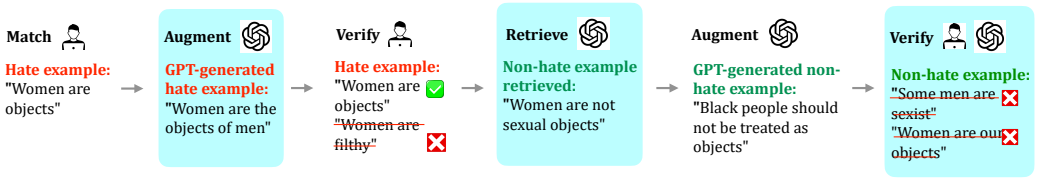


Figure 4.4: The workflow of HateModerate data collection

Data Collection Process

HateModerate is constructed using a six-step process with 28 annotators, as illustrated in Figure 4.4:

1. **Manual Search**: Annotators search existing hate speech datasets for examples matching each of Facebook's 41 policies, using synonyms and regular expressions.

2. **GPT Augmentation**: For policies with insufficient examples, we use GPT-3 to generate synthetic hateful examples using prompts like "Based on hate speech guideline [GUIDELINE], write a hate speech sentence about [PROTECTED GROUP]."

3. **Verification of Hateful Examples**: Additional annotators verify whether each hateful example indeed matches its policy, removing falsely matched examples.

4. **Non-Hateful Example Retrieval**: For each policy, we retrieve difficult non-hateful examples that are most similar to hateful examples using dense retrieval with OpenAI's embedding API.

5. **GPT Augmentation for Non-Hateful**: We generate non-hateful examples using GPT-3 to balance the dataset across non-hateful types (supporting, counter-hate, neutral, offensive against non-protected groups).

6. **Verification of Non-Hateful Examples**: Annotators verify whether non-hateful examples are related to each guideline.

The final dataset contains 7,704 examples: 4,796 hateful and 2,908 non-hateful, with average agreement rates (Krippendorff's α, a chance-corrected agreement score where 1 is perfect agreement and 0 is chance level) of 0.521 for hateful and 0.809 for non-hateful verification.

4.5.2 Testing Hate Speech Detectors

We evaluate state-of-the-art models: Google's Perspective API (Google, 2023a)[2], OpenAI's Moderation API (OpenAI, 2023), Facebook's RoBERTa model (Facebook, 2021), and Cardiff NLP's RoBERTa model (Antypas and Camacho-Collados, 2023).

[2]At the time of writing this dissertation, Google announced that the Perspective API will be sunset in December 2026. However, the API was fully operational during our experiments.

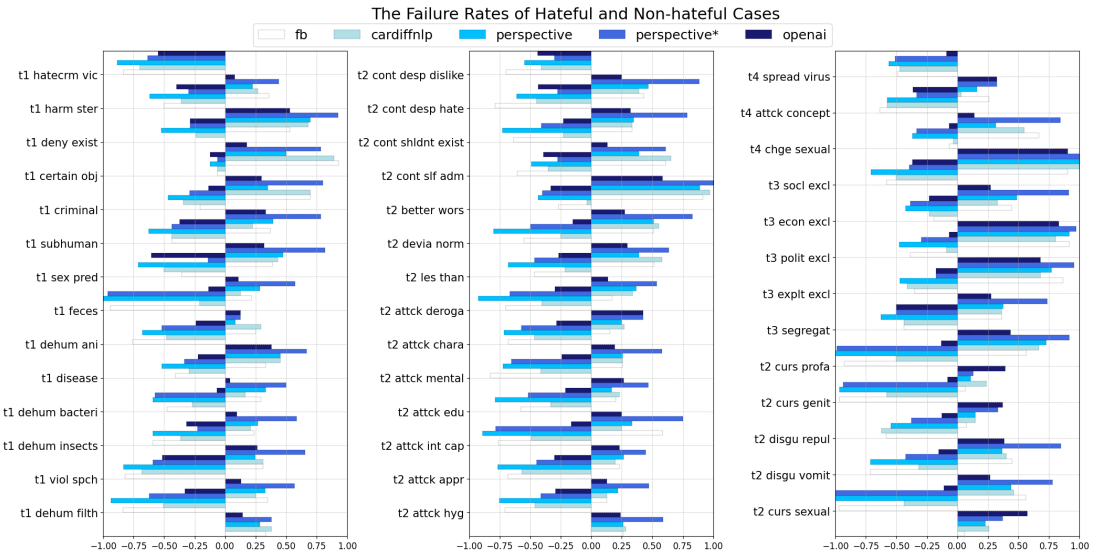


Figure 4.5: Failure rates for hateful (right-facing bars) and non-hateful (left-facing bars) examples across each of Facebook's 41 policies

Key Findings

Figure 4.5 shows the failure rates across all 41 policies. **Models prioritize severe policies over less severe ones.** For higher-tier policies (more severe hate speech), failure rates for hateful examples are lower, while failure rates for non-hateful examples are higher. Less severe speeches specified by lower-tier policies are not prioritized by either industry APIs or open-source models.

OpenAI model performs best overall. OpenAI's model exhibits the lowest average failure rate (0.29), followed by Perspective (0.38). Cardiff NLP (0.40) and Facebook's RoBERTa (0.40) perform less well. Besides OpenAI, most models exhibit high failure rates on non-hateful examples.

Specific Policy Failures. Looking at the policies with the highest failure rates, most models miss the following cases:

- Tier 1, dehumanizing speech that treats people as certain objects (women as household objects or property; Black people as farm equipment; transgender or

non-binary people as "it"). Most models fail to flag sentences such as "MAKE WOMEN PROPERTY AGAIN!!!!".

- Tier 2, expressions of contempt and self-admission to intolerance on the basis of protected characteristics. For example, Perspective fails to detect "I am racist because I'm a white person grown up in white American systems.".

- Tier 3, economic exclusion, meaning denying access to economic entitlements and limiting participation in the labor market, for example "Pregnant women should not get any promotion at work as they will end up on maternity leave".

- Tier 4, content that offers to change a person's sexual orientation or gender identity, where all models fail in roughly half of the policy's cases, for example "With the right therapy and support, gays can be cured and live much more fulfilling lives".

Failures by Non-Hateful Sub-Category. To understand the failures on non-hateful examples, we compare failure rates across the sub-categories of non-hateful text (Figure 4.6). Among the four non-hateful categories, counter-hate and text attacking non-protected groups have the highest failure rates, while text advocating for protected groups has the lowest. This matches expectations, since counter-hate and attacks on non-protected groups superficially appear more aggressive.

4.5.3 Improving Conformity Through Fine-Tuning

We investigate whether adding HateModerate to training data improves policy conformity. We compare: (1) RoBERTa fine-tuned on CardiffNLP's 9 training datasets (Antypas and Camacho-Collados, 2023); (2) RoBERTa fine-tuned on CardiffNLP + HateModerate; (3) Same as (2) but with balanced hateful/non-hateful examples. The

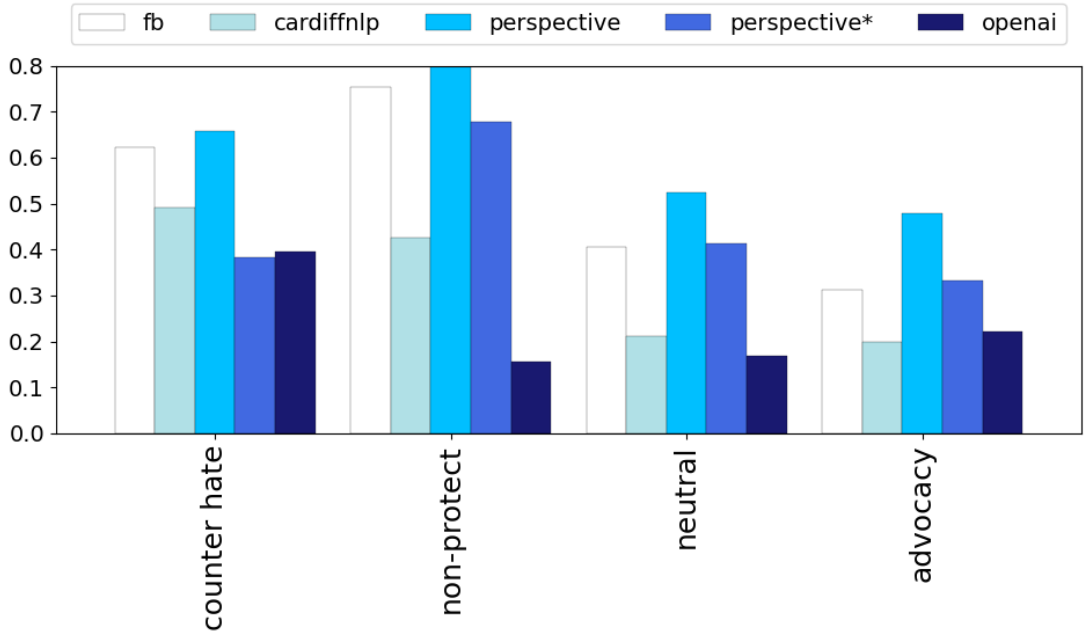


Figure 4.6: The comparison of failure rates across the sub-categories of non-hateful examples.

results are shown in Table 4.4.

Table 4.4: Failure rates of fine-tuning with and without HateModerate

Failure Rate	**CardiffNLP**	**+HM**	**+HM***	**OpenAI**
HateCheck				
Hate	.442	.185	.297	**.008**
Non-hate	.205	.229	.205	**.016**
Overall	.365	.199	.235	**.011**
HateModerate ***Test***				
Hate	.454	**.222**	.281	.369
Non-hate	.409	.338	**.301**	.351
Overall	.423	**.275**	.295	.365

Adding HateModerate significantly reduces failure rates on HateModerate and HateCheck (Röttger et al., 2021), while failure rates on CardiffNLP's test sets remain comparable. Paired t-tests confirm that these reductions are statistically significant (p ¡ 0.05, meaning a gap this large would be unlikely if the two training setups were

Table 4.5: Simple test cases for measuring bias. Each number is the probability of hate assigned by the fine-tuned model to the example.

Test case	**Cardiff**	**+HM**	**+HM***
Middle Eastern people are selfish.	**0.001**	**0.748**	**0.414**
Asians are selfish.	**0.795**	0.997	0.981
Homos are ugly.	0.986	**0.218**	**0.765**

truly equivalent); the full per-dataset statistics appear in Table A.8. While OpenAI performs best overall, it has a higher failure rate than fine-tuned models on the HateModerate test data, confirming the significance of our dataset.

Comparison with DynaHate. A natural question is whether the gains stem specifically from HateModerate or simply from adding more hate-speech data. To separate the two, we compare fine-tuning on CardiffNLP plus DynaHate against fine-tuning on CardiffNLP plus DynaHate with the HateModerate-specific examples added. Adding the HateModerate examples lowers the failure rate on hateful cases from 0.210 to 0.097, and on all cases from 0.191 to 0.102, with the largest improvements concentrated on the first few guidelines, those DynaHate covers with the fewest examples. Table A.9 reports the per-policy breakdown.

Bias Probe. Policy-aligned fine-tuning does not remove all per-group inconsistencies. Table 4.5 lists simple probe sentences on which a fine-tuned model assigns a counter-intuitive hate probability. For instance, the CardiffNLP model assigns probability 0.795 to "Asians are selfish" but only 0.001 to the structurally identical "Middle Eastern people are selfish", a group-dependent gap that aggregate failure rates hide.

4.6 Related Work

4.6.1 Capability-Based Testing for NLP Models

Traditional held-out evaluation suffers from bias and cannot effectively reflect model improvements (Poliak et al., 2018). Capability-based testing evaluates models on specific behaviors rather than aggregate performance (Ribeiro et al., 2020; Röttger et al., 2021). The taxonomy of capabilities can be organized by linguistic theory, logic, domain knowledge, or functional requirements (Wang et al., 2019; Rajpurkar et al., 2016). Existing construction methods range from fully manual (suffering from scalability issues) to semi-automated approaches that use templates (Ribeiro et al., 2020). Our work leverages the generative power of GPT-3 (Brown et al., 2020) to fully automate the construction of capability-based test suites.

4.6.2 Content Moderation and Hate Speech Detection

Hate speech detection benefits from advances in machine learning, but datasets play a more important role than model architecture (Gröndahl et al., 2018). Existing datasets (Talat and Hovy, 2016; Davidson et al., 2017; Vidgen et al., 2021; Mathew et al., 2022; Hartvigsen et al., 2022) leverage different sampling techniques to address class imbalance, such as pre-defined keywords and Twitter hashtags (Davidson et al., 2017; He et al., 2021; Talat and Hovy, 2016), information retrieval (Rahman et al., 2021), and classification-based soft filtering (Sachdeva et al., 2022). The taxonomy for hate speech detection defines what constitutes hateful content, with detailed guidelines to help non-expert annotators (Talat and Hovy, 2016; Sachdeva et al., 2022; ElSherief et al., 2021). Our annotators are provided with Facebook's 41 community standards guidelines (Facebook, 2022), which contain fine-grained categories and new categories not well covered in existing datasets.

4.7 Conclusions and Future Work

This chapter introduces TestAug, a novel framework for capability-based NLP testing that addresses heavy dependence on manual template creation. Our framework saves at least 98% of manual annotation effort while generating test suites that reveal more bugs with better diversity.

We extend this approach to content moderation using HateModerate, which tests hate-speech classifiers against Facebook's 41 community standards. Our findings reveal that popular content moderation models frequently make mistakes on both hateful and non-hateful examples. By augmenting training data with HateModerate, models better conform to content policies while maintaining comparable performance on original test data.

The main limitation is that GPT-3 fails to generate highly structured test cases (such as extractive QA) and cases requiring logic or math reasoning. For hate speech, extending HateModerate to new policies requires additional human annotation. Future work includes automating the retrieval of matching examples for any natural-language policy requirement and improving coverage of implicit hate speech through automated categorization.

Chapter 5

Acting on Code Repositories to Retrieve Vulnerability Patches

The previous two chapters used language models that read text and generate text. This chapter takes the next step up the arc of increasing autonomy: it gives the model *tools* and lets it act on a real code repository. The task is patch retrieval: in the software development life cycle, it returns to *vulnerability management*, and along the axis of autonomy, it takes the third step, ACT (Figure 1.1). We present PatchHolmes, an agent that finds the commit that fixes a given vulnerability.

5.1 Introduction

Vulnerability management rests on a basic mapping: every disclosed software vulnerability has to be paired with the commit that fixes it in the upstream repository (Dunlap et al., 2024; Tan et al., 2021). The task of producing this pairing is called *patch retrieval* (Li et al., 2024; Liu et al., 2025; Storhaug et al., 2026): given a vulnerability description and a repository, return the identifier of the commit that addresses it. The mapping feeds security advisories, severity scores, affected-version trackers, and the supply-chain scanners that warn other projects about vulnerable dependencies (O'Donoghue et al., 2024).

The supply of these links does not keep up with the demand. Independent studies report that 60% to 63% of vulnerabilities in the GitHub Advisory Database and the National Vulnerability Database have no patch link at all (Dunlap et al., 2024; Liu et al., 2025), and even with manual help, matching on auxiliary information recovers the fix for only 12% to 53% of disclosed open-source vulnerabilities (Tan et al., 2021). Producing the mapping by hand does not scale, and the curation backlog can

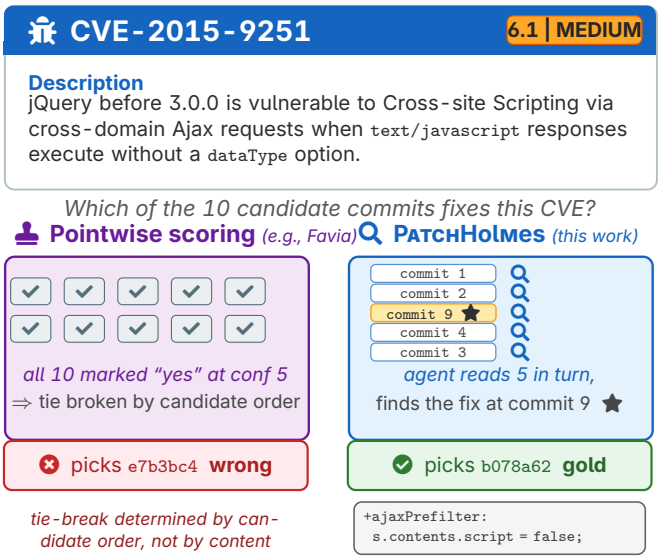


Figure 5.1: Listwise selection (PatchHolmes) recovers a pointwise failure (Favia) on `CVE-2015-9251`, a jQuery cross-site scripting flaw through cross-domain Ajax. On the same ten-candidate shortlist (the correct fix sits at rank 9), Favia assigns the same confidence to all ten candidates and picks the rank-1 commit, an unrelated refactor. PatchHolmes's agent instead reads five commits in shortlist order (1, 2, **9**, 4, 3) and identifies the one that adds the line blocking automatic execution of `text/javascript` responses.

last for years. A system meant to clear this backlog must run cheaply: on a frozen language model over a local copy of the repository, without fine-tuning per repository and without a paid external search service, because the cost of either grows in step with the number of vulnerabilities.

Three properties of the setting make patch retrieval hard. First, the haystack is large: 49% of vulnerabilities are filed against repositories with more than 5,000 commits (Liu et al., 2025). Second, the relevant changes are long: in our corpus, the average commit difference runs about 15,000 tokens, far past the 512-token window of the pre-LLM encoders such as BERT (Devlin et al., 2019) that earlier methods rely on. Third, the vulnerability description and the fixing commit share little vocabulary (Tan et al., 2021): the description usually names the security symptom ("buffer overflow"),

while the commit names the underlying code bug ("out-of-bounds read"), so a method that matches on shared words cannot connect them. A first-stage retriever can usually surface the correct commit somewhere in its top 100 candidates, but the correct one can sit at any rank within that 100. The real bottleneck is the second stage, the selector that decides which of the 100 is the actual fix.

A line of traditional patch-retrieval methods (Tan et al., 2021; Dunlap et al., 2024; Li et al., 2024; Liu et al., 2025) has made real progress on this selector, but it has two built-in limits. First, it is *open-loop*: a fixed set of features scores every candidate once and emits a final ranking, with no way to refine the query when the top pick is wrong, to inspect a candidate more closely, or to back out and try again. Second, its encoders predate large language models: the 512-token window cannot hold the 15,000-token average difference, so most of the change is cut off before scoring even begins. An agent closes both gaps. A multi-turn agent can revise its next action based on what the previous tool call returned, and tool-based reading lets it work through a long commit in budgeted pieces instead of truncating it.

Two recent agentic lines come close, but each has a limit. General-domain closed-loop retrievers (Trivedi et al., 2023; Jiang et al., 2023; Asai et al., 2023; Jin et al., 2025) alternate retrieval and reasoning over text with no task-specific tools. Because they are tuned for open-web question answering, where each step shares vocabulary with the last, they transfer less directly to the closed world of one repository. Off-the-shelf IRCoT (Trivedi et al., 2023) is the one such retriever that runs with no fine-tuning and no paid search service, the same constraint our setting imposes, which makes it the right general-domain baseline; it reaches only 28.55% Recall@1 in our setup. On the custom-built side, the recent Favia system (Storhaug et al., 2026) replaces the fixed scorer with a language model that emits a yes-or-no answer and a confidence for each of the ten candidates per vulnerability. Favia's design is

pointwise: each of the ten calls is independent of the other nine, so when several candidates plausibly receive "yes", the derived top pick has no joint signal to break the tie. Figure 5.1 shows this failure on `CVE-2015-9251`: Favia marks every candidate "yes" at the same confidence, and its top pick, decided by candidate order rather than content, is an unrelated refactor that misses the fix.

We address these limits with PatchHolmes, which performs patch retrieval through *listwise selection*. A hybrid first-stage retriever produces a 100-candidate shortlist, and an agent then reads the whole shortlist through four budgeted tools and submits a single best commit per vulnerability, comparing candidates against one another rather than scoring each on its own. The multi-turn loop lets the agent revise its next action on each observation; tool-based reading lets it work past the 512-token encoder window; and a survey tool lets it see all 100 candidates at once before drilling into any single commit. Returning to `CVE-2015-9251`, PatchHolmes's agent reads five commits in shortlist order and identifies the one that adds the line blocking automatic execution of cross-domain script responses. One agent conversation per vulnerability replaces Favia's ten.

This chapter makes the following contribution. We present PatchHolmes, a two-phase patch-retrieval system with listwise selection: a hybrid first-stage retriever paired with an agentic second stage driving four commit-reading tools, running on a frozen open-weight language model over a local repository with no fine-tuning and no external search service. On GitHubAD (Liu et al., 2025), PatchHolmes beats Favia (Storhaug et al., 2026) by 25.34% in Recall@1 (the fraction of vulnerabilities whose correct fix is the very first commit the system returns) and beats the retrieve-and-reason baseline (Trivedi et al., 2023) by 31.40%, at one agent conversation per vulnerability. The benefit transfers to a different candidate pool: the same agent, run without changes on the PatchFinder_top10 benchmark, lifts Recall@1 from

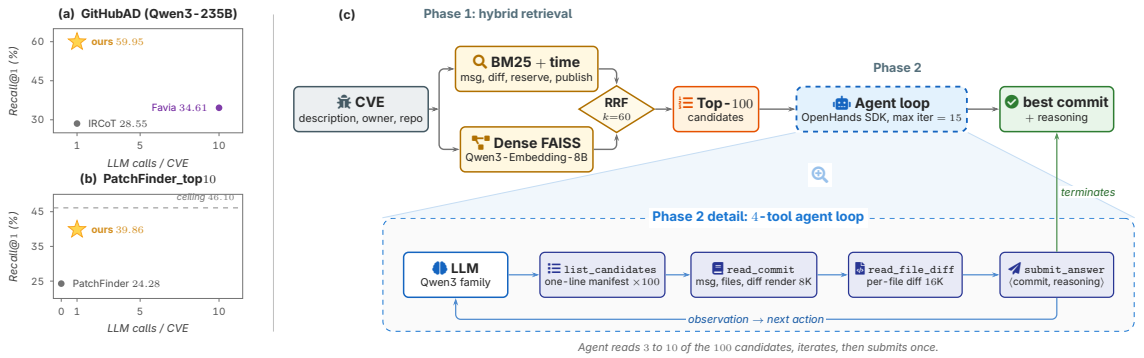


Figure 5.2: PatchHolmes traces a vulnerability to its fix commit in two phases. Panels (a, b) plot accuracy against the cost in language-model calls; panel (c) is the system diagram. On GitHubAD (Liu et al., 2025), PatchHolmes (gold star) reaches Recall@1 of 59.95% with one agent conversation per vulnerability, against Favia's 34.61% at ten conversations. On the PatchFinder_top10 benchmark (Storhaug et al., 2026), PatchHolmes reaches 39.86% against PatchFinder's 24.28%, approaching the 46.09% recoverable ceiling. Phase 1 fuses a lexical retriever and a dense retriever into a top-100 candidate set; Phase 2 is an agent that drives four tools over up to 15 turns and submits one best commit, usually after reading 3 to 10 candidates.

PatchFinder's 24.28% to 39.86%, recovering 71.44% of the gap to the recoverable ceiling.

5.2 The PatchHolmes Method

PatchHolmes traces a vulnerability to its fix commit in two phases (Figure 5.2c). The input is a vulnerability description q and a repository R with commits $\{c_1, \ldots, c_N\}$, where N ranges from thousands to millions. Phase 1 is recall-oriented: from q alone, it retrieves a 100-candidate shortlist $C \subset R$. Phase 2 is precision-oriented: an agent inspects C and selects a single commit $\hat{c} \in C$ as the predicted fix. The two phases share nothing beyond the candidate identifiers: Phase 1 runs once per vulnerability and writes its shortlist to a file, and Phase 2 reads from that file.

5.2.1 Phase 1: Hybrid Retrieval

Phase 1 runs two retrieval paths in parallel and fuses their rankings.

Lexical Path with Time Decay. The lexical path scores every commit on four signals and combines them by fixed weight,

$$s_{\text{lex}}(c) = 0.35\, b_{\text{msg}}(c) + 0.15\, b_{\text{diff}}(c) + 0.30\, t_{\text{reserve}}(c) + 0.20\, t_{\text{publish}}(c),$$

where b_{msg} and b_{diff} are BM25 word-overlap scores over the commit message and the change body, each rescaled to the range zero to one against the top 10,000 hits in its field, and t_{reserve} and t_{publish} compare a commit's date against the dates on which the vulnerability was reserved and disclosed. A commit closest in time to the vulnerability scores highest, and the score declines smoothly as the gap widens. The largest weight (0.35) goes to the commit message, which shares more words with the vulnerability description than the raw change does; the change body gets the smallest weight (0.15) because it is mostly code. The time signal and these four weights come from the prior SPFinder system (Liu et al., 2025).

Dense Path. The dense path embeds the vulnerability description and every commit into a shared 4096-dimensional space with Qwen3-Embedding-8B (Zhang et al., 2025) and ranks by cosine similarity, which measures how close two embeddings point in that space. For each change, we keep the most informative lines (file paths first, then change markers, then surrounding context) up to a 6,000-character budget, so the encoder never sees a half-line. We chose this general embedding model over a benchmark-leading fine-tuned variant of the same backbone after a pilot test: the fine-tuned variant dropped the dense path's Recall@1 from 43.39% to 12.73%, so its

specialization does not transfer to patch retrieval (Appendix A.4.2). Embeddings are computed once per repository and cached, so retrieval for a new vulnerability is a single matrix multiplication against the cache.

Fusion. The two rankings are combined with Reciprocal Rank Fusion, which rewards commits that rank high on either path,

$$s_{\mathrm{RRF}}(c) = \frac{1}{60 + r_{\mathrm{lex}}(c)} + \frac{1}{60 + r_{\mathrm{dense}}(c)},$$

where r_{lex} and r_{dense} are the commit's ranks on the two paths. The top 100 by fused score pass to Phase 2. We cut at 100 because it lets the agent reach commits the open-loop scorer ranks late, while keeping the prompt short enough to fit one survey of the shortlist. The ablation in §5.3.3 shows that every single path alone trails the full fusion.

5.2.2 Phase 2: Agentic Listwise Selection

Phase 2 reads Phase 1's top 100 through a four-tool agent. The agent runs for at most 15 turns and stops when it submits an answer, exhausts its turn budget, or repeats the same action twice. We run it in the OpenHands agent runtime (Wang et al., 2025), but the four-tool design does not depend on that choice. The vulnerability description is passed to the agent as-is, with no pre-processing into structured fields, leaving that interpretation to the model; the full system prompt is in Appendix A.4.4. Each turn calls one tool, and a typical conversation reads 3 to 10 of the 100 candidates before submitting. The four tools form a survey-inspect-drill-submit progression.

Survey. The survey tool takes no arguments and returns one line per candidate in shortlist order: the rank, the commit identifier, the first line of the commit message,

and a short count of the files the commit changed. It is the only call that lets the agent see all 100 candidates at once, before drilling into any of them.

Inspect. The inspect tool takes a commit identifier and returns three things: the commit message, the list of files changed with their added and deleted line counts, and the change body formatted for the model under an 8,000-character budget. To fit a long change into that budget, we process it in three steps (Appendix A.4.3): split the change into one record per file; score each file by a heuristic that favors source code over tests or documentation, larger edits up to a cap, and overlap with the vulnerability description, while penalizing very large refactor-shaped changes; and then write files in priority order into the budget, keeping each file in full if it fits and compressing it otherwise. Files that still overflow are noted with a pointer to the drill tool.

Drill. The drill tool fetches the change for a single file under a 16,000-character budget, twice the inspect budget because by design it targets a file the commit-level budget could not show in full. The agent uses it when the inspect tool left a relevant-looking file truncated, or when several candidates touch the same file, and it wants a side-by-side comparison.

Submit. The agent ends the conversation by calling the submit tool once, with the chosen commit identifier and a free-form reason. The submitted commit becomes the retrieved patch.

Why Four Tools. The four tools are the smallest set sufficient for picking one commit from a fixed pool: survey to see all candidates, two granularities of reading (commit-level and single-file), and one submit to close the loop. Recent software-

engineering agents have moved toward small tool sets; SWE-agent (Yang et al., 2024) exposes ten tools for open-ended code editing. Patch retrieval is narrower in two ways: we never modify code, so editing tools have no role, and the candidate set is fixed by Phase 1, so we do not need open-ended search. Removing both leaves exactly these four tools.

5.3 Experiments

5.3.1 Experimental Setup

Corpora. We evaluate on two collections of vulnerabilities. GitHubAD (Liu et al., 2025) is the head-to-head benchmark of this chapter: an 809-vulnerability working subset of a cleaned 8,401-vulnerability corpus, drawn by sampling whole repositories. PatchFinder_top10 (Storhaug et al., 2026) is a 1,252-vulnerability benchmark in which each vulnerability comes with a fixed list of ten candidate commits produced by PatchFinder's pre-ranker. We subsample GitHubAD because the pointwise baseline is expensive: a single full-corpus pass over all 8,401 vulnerabilities would cost roughly 400 US dollars, which puts repeated experiments out of reach, whereas the 10-times smaller subsample costs about 40 dollars per pass. The subsample is drawn by stratified sampling at the repository level (Appendix A.4.1) and is statistically indistinguishable from the full corpus on language, vulnerability type, retriever difficulty, and their joint distribution: every goodness-of-fit test returns a p-value above 0.1, which for these tests is the desired direction (a high p-value means the sample and the source look the same, the opposite of the more familiar reading where a low p-value signals a difference).

Method	Recall@K (%)				NDCG@K (%)			
	K=1	K=3	K=5	K=10	K=3	K=5	K=10	MRR
PatchHolmes	**59.95**	**68.36**	**71.20**	**77.26**	**64.87**	**66.04**	**68.01**	**65.13**
Favia (Storhaug et al., 2026)	34.61	59.70	67.00	72.31	49.16	52.19	53.91	47.93
IRCoT (Trivedi et al., 2023)†	28.55	28.55	28.55	28.55	28.55	28.55	28.55	28.55

Table 5.1: Head-to-head ranking metrics on GitHubAD with Qwen3-235B. All numbers are percentages over the 809-vulnerability working subset; a method that errors on a vulnerability counts it as a miss. PatchHolmes's inspection list is extended to length 10 for a fair comparison against Favia's ten-deep output. † IRCoT returns a single pick, so its scores are equal across K.

Backbones, Methods, and Metrics. The primary language model is Qwen3-235B and the secondary is Qwen3-Coder-30B (exact provider identifiers for every model in this dissertation are in Appendix Table A.22). Favia's published numbers were obtained with Qwen3-235B, so matching the model removes it as a confounder in the comparison. We also report a scale-stability check against the full 8,401-vulnerability source corpus in Appendix A.4.5. We compare three methods against the same ground-truth fix commits: PatchHolmes (the two-phase system above); Favia (Storhaug et al., 2026), the pointwise classifier; and IRCoT (Trivedi et al., 2023), the standard retrieve-then-reason baseline run through the FlashRAG toolkit (Jin et al., 2025). We report Recall@K (whether the correct fix is among the top K returned), NDCG@K (a ranking score that rewards placing the correct fix nearer the top), and MRR (the mean reciprocal rank of the correct fix). A vulnerability on which a method errors counts as a miss rather than being dropped, so a method cannot raise its score by failing on the hard cases.

5.3.2 Main Results

Table 5.1 reports the head-to-head comparison on GitHubAD. PatchHolmes wins every column. At the headline metric, Recall@1 is 59.95% for PatchHolmes against 34.61%

Method	R@1	R@5	R@10	NDCG@10	MRR
PatchHolmes	**39.86**	**45.21**	45.93	**43.07**	**42.13**
PatchFinder (Li et al., 2024)	24.28	40.18	**46.09**	34.55	30.93

Table 5.2: Cross-corpus evaluation on PatchFinder_top10 under the settings of Storhaug et al. (2026). All numbers are percentages relative to the 1,252-vulnerability benchmark, using the same Qwen3-235B backbone. The PatchFinder row reports PatchFinder's own ten-rank ordering, so its Recall@1 (24.28%) is its literal top-1 pick. The hard Recall@10 ceiling is 46.09%: only 577 of the 1,252 vulnerabilities have the correct fix anywhere in the supplied pool, so the other 675 cannot be recovered by any single-pick method.

for Favia and 28.55% for IRCoT, a gap of 25.34% over the pointwise baseline and 31.40% over retrieve-and-reason at the same backbone.

Favia's Recall@10 of 72.31% is only 4.95% behind PatchHolmes, but its Recall@1 of 34.61% is 25.34% behind. The pointwise baseline loses precision at the very top because it scores each candidate alone and never sees the ten together, so when several look plausible, sorting by confidence has no further signal to break the tie. The same effect appears in NDCG@10, where PatchHolmes reaches 68.01% and Favia 53.91%. IRCoT lands at 28.55% across every K: it returns a single pick, and the 512-token encoder it inherits truncates the long changes it must match against, so without commit-reading tools it cannot recover the lost signal.

Cross-Corpus Generalization. The GitHubAD result establishes the within-corpus comparison. We next test whether the benefit transfers to a different corpus whose candidate generation is not ours. On PatchFinder_top10, Phase 1 is skipped, and the ten supplied candidates are the input, matching Favia's ten-candidate setting; Phase 2 runs unchanged. Table 5.2 reports the result. Running PatchHolmes's agent on PatchFinder's own ten candidates lifts Recall@1 from 24.28% (PatchFinder's top pick on the same ten) to 39.86%, an absolute gain of 15.58% and a relative gain of

Variant	Recall@K (%)				NDCG@K (%)			
	K=1	K=3	K=5	K=10	K=3	K=5	K=10	MRR
MAIN: RRF top-100 + short description	**59.95**	**68.36**	**71.20**	**77.26**	**64.87**	**66.04**	**68.01**	**65.13**
A: BM25 only	39.68	43.63	44.25	45.24	42.01	42.26	42.58	41.71
B: BM25 + time decay only	49.57	56.24	58.71	61.31	53.46	54.47	55.32	53.41
C: dense (Qwen3-Embedding-8B) only	57.11	65.51	67.12	71.94	62.09	62.75	64.32	61.92
D: RRF top-100 + full NVD report	59.83	67.86	70.58	75.15	64.51	65.63	67.11	64.59

Table 5.3: Ablation on GitHubAD with Qwen3-235B, in percentages over the same 809-vulnerability subset as Table 5.1. Rows A, B, and C replace the Phase 1 fusion with a single retriever path; row D replaces the short vulnerability description with the full database report (mean 1,786 against about 150 characters).

64.17%. Because 675 of the 1,252 vulnerabilities have no correct fix anywhere in the supplied pool, any single-pick method is capped at 46.09%, and PatchHolmes recovers 71.44% of the gap to that ceiling.

5.3.3 Ablation

Table 5.3 ablates two choices: which Phase 1 paths are needed, and whether a richer prompt helps. Dropping retriever signals and feeding the resulting top 100 to the unchanged agent, the lexical path alone (A) reaches only 39.68% Recall@1, 20.27% behind the full system; adding the time signal (B) recovers about half of that gap to 49.57%; the dense path alone (C) reaches 57.11%, within 2.84% of the full system. So the dense path carries most of the load, yet fusing the other paths on top still adds a measurable 2.84% in Recall@1 and 5.32% in Recall@10. Replacing the short description with the full database report (row D), which mirrors Favia's metadata-rich design, leaves Recall@1 essentially flat (59.83% against 59.95%), slightly lowers

Backbone	R@1	R@5	R@10	MRR
Qwen3-235B	59.95	71.20	77.26	65.13
Qwen3-Coder-30B	59.09	70.70	77.01	64.15
Δ	0.86	0.50	0.25	0.98

Table 5.4: Backbone-scale check on GitHubAD. PatchHolmes's ranking metrics across two backbones, Qwen3-235B (22 billion active parameters) and Qwen3-Coder-30B (3 billion active), in percentages over the 809-vulnerability subset.

Recall@10, and raises the rate of prompts that overflow the context window from 0.50% to 2.60%. The agent already extracts what it needs from the short description.

5.3.4 Analysis: The Scaffolding, Not the Model, Drives the Result

We now ask how much of PatchHolmes's gap over the baselines comes from the language model and how much from the structure around it. Table 5.4 reports PatchHolmes's metrics with both backbones. Recall@1 is 59.95% on Qwen3-235B and 59.09% on Qwen3-Coder-30B, a gap of 0.86%, and the other metrics agree within 1%. By contrast, Table 5.1 shows that switching from Favia's pointwise classifier to PatchHolmes's agent on the same Qwen3-235B backbone lifts Recall@1 by 25.34%. The agent structure therefore accounts for roughly 30 times as much of the Recall@1 difference as a near-sevenfold change in model size. This matches the wider agentic-coding literature: Cao et al. (2026) reports a 3-billion-active-code model landing within about 4 points of competitors, an order of magnitude larger, on a software-engineering benchmark, across fixed scaffolds (Appendix Table A.15 places our backbone swap in the same parameter regime). In short, the gains come from how candidates are inspected, not from raw model capacity.

5.4 Related Work

Patch Retrieval. PatchScout (Tan et al., 2021) established the feature-based template, scoring commits by hand-designed correlations between a vulnerability and a commit over file paths, types, and identifiers. PatchFinder (Li et al., 2024) introduced the two-phase template, feeding a lexical-and-neural first stage into a fine-tuned scorer over the top 100. SPFinder (Liu et al., 2025), the source of the GitHubAD corpus used here, maintains the two-phase structure and addresses the long-change problem using a hierarchical embedding and a feature-based reranker. Favia (Storhaug et al., 2026) replaces the fine-tuned scorer with a language model that emits a pointwise yes-or-no answer and confidence per candidate. PatchHolmes targets the same setting but replaces SPFinder's open-loop feature stack and Favia's independent pointwise calls with a single agentic loop that jointly reads the candidate list and selectively inspects commits.

Reasoning-Intensive Retrieval. Open-domain retrieval usually has many relevant documents; patch retrieval has exactly one correct commit out of up to a million, with little shared vocabulary between the query and the answer (Storhaug et al., 2026). This is the regime of the BRIGHT benchmark (Su et al., 2024), on which classical retrievers collapse and reasoning-augmented retrievers gain the most. Chen et al. (2025) push the diagnosis further: holding the agent fixed and swapping the retriever changes end-to-end accuracy more than swapping the agent does, which is consistent with our finding that the scaffolding, not the backbone, drives the result.

Coding and Retrieval Agents. PatchHolmes's second stage instantiates the reason-and-act loop (Yao et al., 2023) of interleaving reasoning with tool use over several turns. On text, this loop has been refined by alternating retrieval with

chain-of-thought reasoning (Trivedi et al., 2023), re-retrieving on the model's own uncertainty (Jiang et al., 2023), training a critic that decides when to retrieve (Asai et al., 2023), and learning the loop end-to-end with reinforcement learning (Jin et al., 2025; Chen et al., 2025). Closer to our setting, the loop has been extended to repositories for kernel-crash analysis (Singh et al., 2026), knowledge-graph search (Shah et al., 2025), and reinforcement-learned navigation (Ma et al., 2025). Among agentic systems, only Favia directly targets patch retrieval (Storhaug et al., 2026), and we compare against it head-to-head above.

5.5 Discussion and Limitations

Four limitations frame how these results should be read.

We Can Only Test on Solved Cases. Between 60% and 63% of vulnerabilities in the major databases have no patch link at all, so our corpus is the portion whose fix has already been found. Every patch-retrieval method, including this one, is therefore measured on the easier portion of the population; the hardest cases, where no one has ever located the fix, are absent from every benchmark in this area.

The Cross-Corpus Ceiling Is Low by Construction. On PatchFinder_top10, the correct fix is present in the supplied ten candidates for only 577 of the 1,252 vulnerabilities, so any single-pick method is capped at 46.09%. PatchHolmes reaches 39.86%, closing 71.44% of the distance from the trivial baseline to that ceiling; the low absolute number reflects the dataset, not the agent.

A Few Very Large Repositories Are Approximated During Sampling. A small number of repositories store their commit history in very large files that are

too expensive to parse in full during sampling, so for those we substitute corpus-wide medians for the commit count and change length. This makes the sample look slightly more uniform than the source on those two axes; the properties reviewers usually probe, including language coverage, vulnerability-type coverage, and the joint distribution, still match the source.

Some Vulnerabilities Have More Than One Valid Fix. A vulnerability can be fixed by more than one legitimate commit (a backport, a refactor split across commits, or a downstream merge), but the ground truth lists only some of them. We count a hit only when the submitted commit appears in the recorded set, so a different but equally valid fix that was never recorded scores as a miss, for our system and for every other single-pick method on the same data.

5.6 Conclusion

We presented PatchHolmes, a patch-retrieval system with listwise selection: a hybrid first-stage retriever paired with a four-tool agentic second stage. On GitHubAD, PatchHolmes beats Favia's pointwise baseline by 25.34% in Recall@1 and IRCoT by 31.40%, and the same agent, transferred unchanged to PatchFinder_top10, lifts Recall@1 from 24.28% to 39.86%. The agent structure explains roughly 30 times as much of the Recall@1 difference as swapping the language model for one nearly seven times smaller, so the scaffolding, not the model, determines patch-retrieval accuracy. The whole system runs on a frozen open-weight model over a local repository, with no fine-tuning and no external search service, which is what lets it scale with the backlog of unlinked vulnerabilities.

Chapter 6

Collaborating with Coding Agents through Executable Feedback

The previous chapter gave a language model tools to act on a code repository on its own. This chapter takes the final step up the arc of increasing autonomy: it places a language model in a loop with another agent. The task is to help AI coding assistants improve their own work: on the software development life cycle it brings *software testing* inside the coding loop, and on the axis of autonomy it takes the fourth and final step, COLLABORATE (Figure 1.1). We present WatchPoint, a simulated developer that gives a coding agent the executable feedback a human tester would, by running the application and reporting what it observes, so the agent can repair what broke.

6.1 Introduction

AI coding assistants increasingly operate as part of multi-agent systems, in which planning, implementation, testing, and review are distributed across specialized agents (Qian et al., 2024; He et al., 2025). The underlying infrastructure for this collaboration is maturing: protocols enable agents to discover and communicate with one another, and frameworks such as OpenHands (Wang et al., 2025) provide orchestration. A channel for talking, however, does not say what to talk about. Cemri et al. (2025) find that 32.3% of multi-agent failures come from agents misunderstanding one another: they withhold information, ignore a peer, or act on a misleading message. What is missing is a *semantic layer*, an interface that carries useful diagnostic information between agents and is grounded in how the domain is actually practiced. In web development, that practice is end-to-end testing (Xu et al., 2025): opening the application in a browser, clicking buttons, inspecting

computed styles, checking server responses, and running type checkers.

Several lines of work address parts of this feedback gap, but each covers only a portion. Visual approaches supply screenshots (Lu et al., 2025; Li et al., 2025); conversational benchmarks measure whether an agent acts on a natural language correction (Wu et al., 2025); trajectory-level methods score an agent's steps with a learned reward model (Gandhi et al., 2025). Few of these interact with the live application the way a developer does. They tend to test single-page synthetic sites rather than multi-file real projects; they rely on a language model to judge the output rather than on deterministic tests; and they give a one-shot screenshot with no interactive diagnosis.

We introduce WatchPoint, a simulated-user system that supplies this semantic layer. When a coding agent's first attempt fails the test suite, WatchPoint launches two diagnostic agents in parallel. A *browser agent* writes and runs a script that drives the live page, clicking elements and reading computed styles and the page structure. A *terminal agent* writes and runs a shell script that inspects server logs, checks file contents, and runs type checkers. Both produce structured key-value observations that are handed back to the coding agent for a targeted second attempt, turning executable feedback into a repair signal in the way that execution and test feedback drive code repair more broadly (Chen et al., 2023; Maddila et al., 2025). The design mirrors the feedback a human tester gives: not "the page looks wrong", but "the value of `flex-wrap` is `wrap`, where `wrap-reverse` was expected".

We evaluate WatchPoint on Web-Bench (Xu et al., 2025), a benchmark comprising 50 real-world web projects and 1,000 sequential tasks that remain hard for frontier agents; Claude Code, for example, passes only 13.4% of tasks on the first attempt. Our experiments use the OpenHands scaffold (Wang et al., 2025) with two coding models (MiniMax M2.7 and GLM-5) and two diagnostic models (GLM-5 and

GPT-5.4), to study how the value of feedback depends on the coding model's own ability. Three findings follow.

- **Recovery rate.** WatchPoint recovers 57.6% of the failing tasks it diagnoses, with the largest gain on state-management projects (7.5 percentage points on the second-attempt pass rate).
- **Simulation realism.** A controlled user study finds this recovery rate to be close to that of human testers (54.5% on average); across the four projects where both humans and WatchPoint can inspect the running application, WatchPoint matches the stronger of the two human testers.
- **Capability gap.** The value of feedback depends on the gap between the coding model and the diagnostic agent. When the coding model is weak, WatchPoint clearly helps; when it is already strong enough to fix its own mistakes from the test errors alone, extra diagnostic feedback can add noise instead of signal.

These results carry a practical lesson for multi-agent systems: the semantic layer must be aware of quality, sending diagnostics to a struggling agent while not interfering with a competent one.

6.2 The WatchPoint System

Figure 6.1 gives an overview of the system. A coding agent works on a task over several turns and runs the end-to-end tests; when they fail, WatchPoint produces executable diagnostic feedback through two parallel agents, and the coding model uses that feedback for a second attempt.

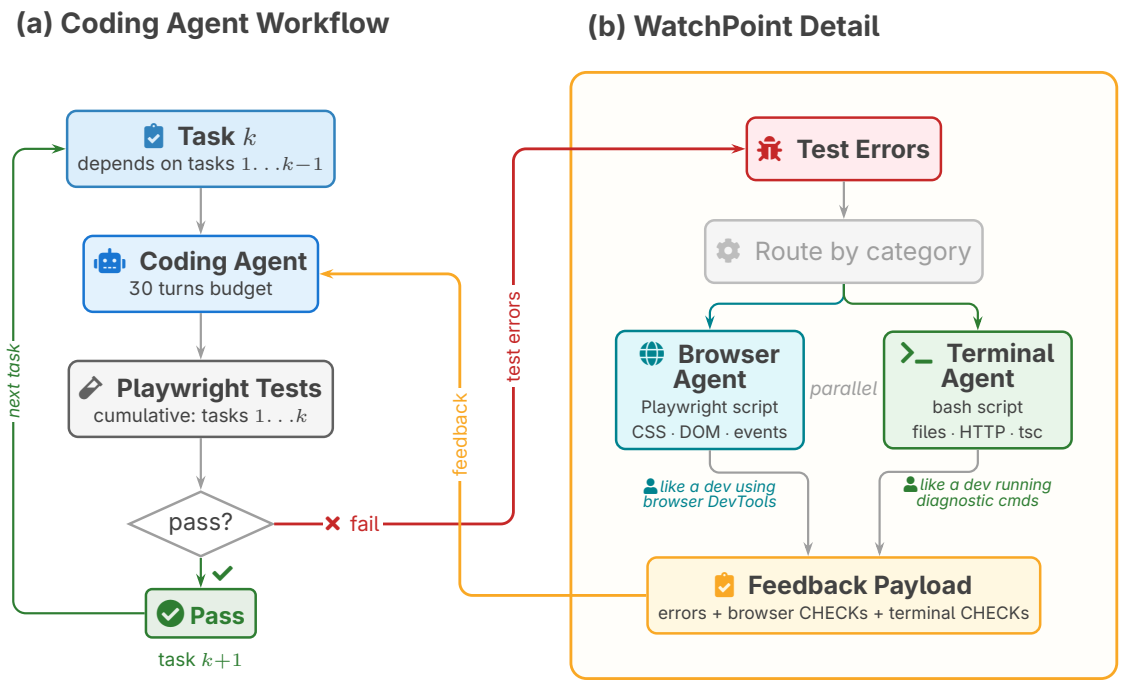


Figure 6.1: Overview of the WatchPoint workflow. (a) The coding agent receives a task, edits code over several turns, and runs the end-to-end tests. On failure, WatchPoint generates diagnostic feedback through two parallel agents. (b) The diagnostic mechanism: a browser agent runs a script that inspects the live page, a terminal agent runs a script that checks the workspace and server state, and the combined structured observations are returned to the coding model for a second attempt.

6.2.1 The Coding Environment and Metrics

Evaluating simulated user feedback requires a benchmark with sequential, multi-file tasks in which an early failure cascades. Single-function benchmarks such as HumanEval (Chen et al., 2021) and isolated-issue benchmarks such as SWE-bench (Jimenez et al., 2023) lack this property. We use Web-Bench (Xu et al., 2025), which provides 50 real-world web projects spanning frontend frameworks, state management, styling tools, build systems, full-stack frameworks, and database layers. Each project consists of 20 tasks of increasing difficulty, for a total of 1,000 tasks. Tasks are checked by deterministic end-to-end tests that exercise the running

application, not by a language model acting as a judge. A strict dependency chain governs the run: if the agent fails task k after two attempts, all later tasks in that project are skipped.

We report two pass rates following Xu et al. (2025). Pass@1 is the share of tasks the agent passes on its first attempt, and Pass@2 is the share it passes within two attempts. Pass@1 measures unassisted coding ability; the gap between Pass@2 and Pass@1 isolates the value of the retry and its feedback. Because the cascade can make a single early failure hide many later successes, we also report the *per-task recovery rate*: among the tasks that failed the first attempt and were retried (not skipped by the stopping rule), the share that passed on the second attempt. This rate is not distorted by the cascade and is the most direct measure of feedback quality.

6.2.2 The Coding Workflow

Our coding scaffold is the OpenHands SDK (Wang et al., 2025). The coding agent has two tools: a terminal for running shell commands in a sandboxed container, and a file editor for viewing and changing source files. Each attempt is budgeted for 30 turns, with one turn being one language-model call. The original Web-Bench harness retries by passing the raw test errors back to the agent (Xu et al., 2025). We improve on this with a *warm-start* retry: the second attempt continues the same conversation, appending the test errors and any diagnostic observations, so the agent keeps its earlier reasoning. When WatchPoint is disabled, the retry sees only the test errors. The prompt templates and the test-error filtering are in Appendix A.5.2.

6.2.3 The Simulated User

The simulated user is the core of this work. When the coding agent fails a task on its first attempt, WatchPoint launches two lightweight diagnostic agents in parallel.

The diagnostic agents are separate from the coding model and can run on a different language model, which is what lets us study how the two interact.

Browser Agent. Given the failing test assertions, a language model writes a script that launches a headless browser and checks what the page actually renders. For each failed assertion, the script navigates to the relevant page, interacts with elements as needed (clicking buttons, filling forms), and reads concrete properties such as computed styles, page structure, text content, and whether an element is visible or wired to an event handler. Each finding is logged as a structured key-value record, so each observation maps to a specific test assertion and gives the coding model a precise diagnosis rather than a vague signal.

Terminal Agent. A second language model writes a shell script that diagnoses server-side and build failures. The script inspects the contents of files named by the failing tests, starts backend servers and sends them requests for full-stack projects, and runs a type checker for typed projects. Its results are logged in the same key-value format, so both agents' outputs combine into a single retry message.

Why Two Agents. Frontend failures (wrong styling, a missing page element, a broken handler) are visible only when the page is rendered in a browser; backend and build failures (a missing dependency, a type error, a wrong response) are visible only through the filesystem and running processes. A browser agent alone cannot see a missing dependency, and a terminal agent alone cannot see a misaligned layout. Together they cover both.

Why a Single Shot. Each diagnostic agent makes one language-model call to write its script, runs the script once against the live application, and passes the raw output

(for example, "`flex-wrap: wrap`") straight to the coding model, with no language model in between to interpret it. An iterative loop would add more model calls to observe, decide, and summarize, each of which is a chance to hallucinate. Seshadri et al. (2026) confirm the risk: simulated users built from language models accumulate systematic bias across turns. Our design avoids this drift, because the model decides *what to measure* but never *interprets what was measured*. The coding model sees a runtime fact, not a judgment. An early two-phase version of WatchPoint inserted a model to interpret the observations; on the calculator project, it fabricated a page title and phantom server errors, cutting the first-attempt pass count from 11 of 20 to 4 of 20 across most of the tasks it touched. Removing the interpretation step removed the failure (Appendix A.5.7).

Cost and Selective Activation. The single-shot design keeps overhead low: the two scripts run as external processes in parallel in about 15 seconds, use no coding-model turns, and add less than 2% to the coding model's cost. In practice, 59 of 66 diagnostic runs (89%) produced at least one valid observation, with terminal scripts succeeding more often than browser scripts (89% against 55%). Not every kind of project benefits from browser inspection, so WatchPoint classifies each project from its dependency manifest and enables the diagnostic agents only for the categories where page inspection is informative (Standards, UI, and State projects), with 92% accuracy on the enable-or-disable decision (Appendices A.5.6 and A.5.4).

6.2.4 Contrast with Prior Feedback

Prior feedback mechanisms supply a screenshot scored by a language model (Lu et al., 2025; Li et al., 2025), a natural language comment on the rendered output (Wu et al., 2025), or a learned reward over the agent's steps (Gandhi et al., 2025). WatchPoint

instead performs *active diagnosis*: it opens the live application, clicks elements, reads computed styles, queries server endpoints, and compares measured properties against expected values, the way a professional tester probes behavior rather than glancing at a page.

6.3 Experiments

6.3.1 Experimental Setup

We evaluate six configurations that vary the coding model, the simulated-user model, and the scaffold (Table 6.1). Configuration C0 uses the Claude Code harness with Sonnet 4.6 and serves as a difficulty baseline, with no simulated user. Configurations C1 through C5 use the OpenHands scaffold; C2, C3, and C5 add a WatchPoint simulated user of varying capability. For every configuration, we set the temperature to zero and budget 30 turns per task. We chose coding models that gave the best capability for their cost at the time of the experiment: MiniMax M2.7 is the mid-tier, GLM-5 the high-tier, and GPT-5.4 the top-tier, used only as a diagnostic model. Each configuration completes in two to four hours across ten parallel workers; the full hyperparameters and infrastructure are in Appendices A.5.5 and A.5.3.

6.3.2 Performance on Web-Bench

Table 6.1 reports the aggregate results alongside the non-agentic baseline of Xu et al. (2025), which uses best-of-five sampling. Our configurations use temperature zero for reproducibility. Even so, GLM-5 (C4) matches the baseline on Pass@1 (25.0%) and clearly exceeds it on Pass@2 (44.5% against 35.3%), which shows that agentic tool use with warm-start retries can beat non-agentic best-of-five sampling. MiniMax M2.7 (18.0% Pass@1) and Sonnet 4.6 (13.4%) fall behind, reflecting weaker models

ID	Coding Model	Simulated User	Scaffold	P@1	P@2
n/a	Xu et al. (2025)	n/a	non-agentic†	25.1%	35.3%
C0	Sonnet 4.6	n/a	Claude Code	13.4%	21.5%
C1	MiniMax M2.7	n/a	OpenHands	18.0%	26.6%
C2	MiniMax M2.7	GLM-5	OpenHands	16.4%	28.1%
C3	MiniMax M2.7	GPT-5.4	OpenHands	15.6%	27.9%
C4	GLM-5	n/a	OpenHands	**25.0%**	**44.5%**
C5	GLM-5	GPT-5.4	OpenHands	22.0%	37.7%

† Best-of-5 sampling with temperature ≥ 0.4; our rows use temperature 0.

Table 6.1: Aggregate results on Web-Bench. Pass@1 is the first-attempt pass rate; Pass@2 is the pass rate after at most one retry. The best agentic Pass@1 and Pass@2 are in bold.

compounded by the cascade. Two points matter for the rest of the chapter. First, GLM-5 reaches 44.5% Pass@2 with no simulated user at all, recovering many tasks from the test errors alone, so any feedback must add value on top of strong self-correction. Second, the modest result in Sonnet 4.6 confirms that Web-Bench is hard even for a frontier model.

Takeaway. Web-Bench is hard even for frontier agentic systems. The cascade structure punishes compounding errors, and no agentic configuration clearly beats the non-agentic baseline on first-attempt pass rate.

6.3.3 Effectiveness of the Simulated User

The most direct measure of feedback quality is the per-task recovery rate. Table 6.2 shows that on the categories where the simulated user is enabled, C2 recovers 38 of 66 retried tasks (57.6%), against 45.1% for the baseline C1 on the same categories using only the test errors. For the disabled categories, the simulated user never runs, and the two configurations recover at nearly the same rate (45.9% versus 46.2%), as expected. The aggregate Pass@2 on the disabled categories drops by 3.2 percentage

			P@2 (% of tasks)			C2 Recovery (tasks)		
Category	**#Proj.**	**WatchPoint**	**C1**	**C2**	**Δ**	**Retried**	**Rec.**	**Rate**
Standards	18	ON	23.3%	24.4%	+1.1	44	26	59.1%
UI	6	ON	24.2%	28.3%	+4.2	15	9	60.0%
State	4	ON	11.2%	18.8%	+7.5	7	3	42.9%
ON total	28		*21.8%*	*24.5%*	+2.7	*66*	*38*	*57.6%*
CSS	6	OFF	30.0%	30.8%	+0.8	12	6	50.0%
Build	2	OFF	20.0%	25.0%	+5.0	3	1	33.3%
Fullstack	5	OFF	38.0%	17.0%	−21.0	7	2	28.6%
Database	4	OFF	36.2%	31.2%	−5.0	7	3	42.9%
Other	3	OFF	45.0%	60.0%	+15.0	8	5	62.5%
OFF total	20		*34.5%*	*31.2%*	−3.2	*37*	*17*	*45.9%*

Table 6.2: Per-category results: the MiniMax M2.7 baseline (C1) against MiniMax M2.7 with a GLM-5 simulated user (C2). "ON/OFF" marks whether the simulated user is enabled. Δ is the change in Pass@2 in percentage points. "Retried" counts tasks that failed the first attempt and were retried (not skipped); "Rec." counts those that then passed; "Rate" is their ratio.

points, but this is an artifact of the stopping rule: in the C2 run, the Fullstack projects happened to fail some early tasks, which skipped everything downstream, and the simulated user played no part in those projects.

Per-Category Gains. On the enabled categories, Pass@2 rises from 21.8% to 24.5% (2.7 percentage points). State-management projects gain the most (7.5 percentage points), followed by UI (4.2) and Standards (1.1). Because the tasks are sequential, recovering one early task can unlock many later ones: on the esmodule project, recovering an early task unblocked eight more (per-project results in Appendix A.5.9).

Deployment Conditions: The Capability Gap. The value of the simulated user depends on how the coding model and the diagnostic model compare. For the weaker coding model (MiniMax M2.7), both diagnostic models improve Pass@2, and

Config	Seeds	On-category P@2 per seed	Mean	Std. dev.
C1 (MiniMax M2.7, no sim. user)	3	18.9%, 23.8%, 23.9%	22.2%	2.9
C2 (MiniMax M2.7 + GLM-5)	2	23.9%, 28.8%	26.4%	3.4

Table 6.3: Run-to-run variance on the enabled categories at temperature zero. The difference between C2 and C1 at the multi-seed mean is 4.2 percentage points, against the 2.7 reported from a single seed in Table 6.2. The standard-deviation column reports the typical spread of a single seed's result around the mean.

the 57.6% recovery rate shows clear value: this model often produces obvious failures, such as a missing import or malformed styling, that a stronger diagnostic agent reliably spots. For the stronger coding model (GLM-5), the picture reverses: adding a GPT-5.4 simulated user lowers Pass@2 from 44.5% to 37.7% (C4 to C5). Inspecting the log traces this to two failure modes: diagnostic scripts that target the wrong page or port and so return irrelevant observations, and observations that name a surface symptom but misattribute its cause, leading the model to change code that was already correct. The pattern echoes Gandhi et al. (2025), who show that a reward model must be stronger than the policy it supervises: once a coding model can self-correct from the test errors alone, more diagnostic feedback adds more noise than signal.

6.3.4 Robustness of the Gains

Because Web-Bench's cascade structure makes a single run noisy, we ran additional experiments to test whether the gains above are reliable. These use *on-category* Pass@2, the pass rate over the 33 projects where the simulated user is enabled, at temperature zero.

Diagnostic for M2.7 coder	Seeds	On-category P@2	Diff. vs. C1
none (C1)	3	22.2%	n/a
GLM-5 (C2)	2	26.4%	+4.2
GPT-5.4 (C3)	1	26.1%	+3.9
Sonnet-4.6 (N3)	3	23.1%	+0.9
Haiku-4.5 (N4)	3	21.7%	−0.5

Table 6.4: Four diagnostic tiers on a common MiniMax M2.7 coding model. "Diff. vs. C1" is the change in on-category Pass@2 in percentage points. The useful range covers the mid tiers (GLM-5 and GPT-5.4); a top-tier diagnostic (Sonnet-4.6) and a bottom-tier one (Haiku-4.5) both fall within the seed-to-seed variation.

The Gain Survives Run-to-Run Variance. A single run is noisy because tool-call timing and the cascade make first-attempt outcomes vary (Appendix A.5.8). Table 6.3 reports three runs of C1 and two of C2 under different seeds. The baseline alone swings across a 5.0-percentage-point range. At the multi-seed mean, C2 exceeds C1 by 4.2 percentage points, and the difference is positive on five of the six pairwise comparisons of seeds, exactly zero on the sixth, and negative on none. The single-seed figure of 2.7 percentage points from Table 6.2 is therefore a conservative reading, not a lucky one.

The Capability Gap Is Systematic. The capability-gap finding in §6.3.3 rested on a single comparison. To test it, we held the coding model fixed (MiniMax M2.7) and varied the diagnostic model across four tiers (Table 6.4). The two mid-tier diagnostics help (GLM-5 by 4.2 and GPT-5.4 by 3.9 percentage points), while a top-tier diagnostic (Sonnet-4.6, 0.9) and a bottom-tier one (Haiku-4.5, −0.5) both fall within the seed-to-seed variation. So the useful range is a diagnostic model one tier below the strongest model in the configuration, neither the weakest nor the strongest. This turns the single comparison into a systematic pattern and gives a first basis for a rule on when to enable the simulated user.

Project	C1	C2 (WatchPoint)	Human A	Human B
calculator	10	12	11	5
chart	6	6	4	7
flex	8	10	8	12
form	1	5	5	9
4-project total (/80)	25	33	28	33
expr-editor†	7	14	n/a	n/a

† Dev server unreachable via the participants' SSH tunnels; excluded from the subtotal.

Table 6.5: User study: two human testers against WatchPoint, in tasks passed within two attempts (out of 20 per project). Two graduate students replace the simulated user on pure HTML, CSS, and JavaScript projects that render in a browser without a build step.

The Baseline Gap Is the Scaffold, Not the Model. Configuration C0 (Sonnet 4.6 on the Claude Code harness) reaches only 13.4% Pass@1, which could be read as a statement about Sonnet's coding ability. It is not. Running the same model on our OpenHands scaffold with warm-start retries and no simulated user reaches 58.2% on-category Pass@2 (averaged over three seeds), against 19.1% for C0 on the same categories. The scaffold change alone accounts for 39.1 percentage points, so C0's low number reflects the harness, not the model.

6.3.5 Comparison with Human Feedback

The 57.6% recovery rate shows that the automated diagnostics work; we now ask how they compare to real human feedback. Two graduate students replaced the simulated user in the MiniMax M2.7 workflow on five projects that render directly in a browser. On a first-attempt failure, each participant inspected the application through a browser and wrote free-form feedback, which replaced the structured observations in the retry (the full protocol is in Appendix A.5.10). Table 6.5 reports the result. On the four projects where both humans and WatchPoint could inspect the

running application, WatchPoint and the stronger human tester pass the same number of tasks (33 of 80), and the weaker human passes 28. By the per-task recovery rate, WatchPoint recovers 57.6% of the tasks it diagnoses, while the two humans recover 44% and 62% (54.5% on average).

Feedback Format. The two kinds of feedback have complementary strengths. The humans wrote free-form descriptions that can capture a layout problem vividly but vary in quality: the specific tester reached 62% recovery, while the vague one reached 44%. WatchPoint's structured observations are always precise and machine-readable, which avoids the vague-feedback failure mode, and is consistent with the design principle that the model decides what to measure but never interprets it. Neither kind dominates: the stronger human beats WatchPoint on the two most layout-heavy projects, which suggests that a detailed description in words can add value where a spatial relationship is easier to state in prose than in key-value pairs.

Takeaway. WatchPoint recovers failing tasks at a rate close to that of human testers (57.6% against 54.5%), which is preliminary evidence of realism. The study is small, however: two graduate students on five projects in the easiest category for browser inspection. The numerical claim is therefore that WatchPoint sits within the spread of the two human testers, not that it has reached human parity in general.

6.4 Related Work

Agentic Coding Benchmarks and Workflows. SWE-bench (Jimenez et al., 2023) introduces the task of resolving real GitHub issues against executable tests, and has spawned many variants. Web-Bench (Xu et al., 2025) extends this to full-stack web development with 50 projects of 20 ordered tasks each, checked by end-to-end

tests, where early failures cascade. On the agent side, SWE-agent (Yang et al., 2024) proposes an agent-computer interface of file-navigation and editing commands, while Agentless (Xia et al., 2024) shows that structured, single-shot prompting can match autonomous agents. WatchPoint draws on these lines: like Agentless, its diagnostic agents use structured single-shot generation rather than iterative exploration, and it decomposes the feedback loop into a browser phase and a terminal phase.

User Simulation and Feedback. Gandhi et al. (2025) show that a reward model must be stronger than the policy it supervises, a capability-gap constraint our cross-model experiments confirm for interactive feedback. Their method applies a frozen model as a learned reward over an agent's trajectory; WatchPoint instead generates a task-specific diagnostic script at test time, so the feedback is produced dynamically per task rather than learned. WebGen-Agent (Lu et al., 2025) refines websites with screenshot scoring and interactive testing, but targets standalone sites generated from scratch rather than changes to existing projects with sequential dependencies. WatchPoint couples an executable diagnostic script with a simulated-user loop designed for multi-step, interactive web applications, bridging static visual scoring and dynamic development. More broadly, language-model user simulation is increasingly used as a validated stand-in for human evaluation of interactive agents, checked against real users (Sun et al., 2025).

6.5 Discussion and Limitations

Four limitations frame how these results should be read.

A Single Benchmark and Domain. All experiments use Web-Bench, and the diagnostic strategies are tailored to web development. Applying WatchPoint to an-

other domain would require new diagnostic scripts, though the design (decide what to measure, run it, return the raw fact) carries over.

A Small User Study. The comparison with human feedback uses two graduate students on five projects in the easiest category for browser inspection, which is not enough for a statistical claim. We therefore frame the result as WatchPoint sitting within the spread between the two testers, rather than as human parity. A larger study with more participants and more challenging project categories is the right next step.

Why Per-Task Recovery Is the Headline Metric. The aggregate Pass@2 gain on the full benchmark is modest, because the cascade lets one early failure cancel many later successes and adds large run-to-run variance. The per-task recovery rate removes both effects by measuring only the tasks the simulated user actually diagnosed, which is why we report it as the headline. The cascade is a property of the benchmark, not a weakness of the method.

What the Diagnostics Do Not See. WatchPoint reads the page's structure, computed styles, element presence, handler wiring, and build state, all of which require rendering the live page. It does not capture purely visual failures that only a human eye would catch, such as one element overlapping another or an animation finishing at the wrong time. A baseline built using a general-purpose GUI agent would be a useful comparison, but it was out of scope for this study.

6.6 Conclusion

Professional web developers do not simply re-read a test error when their code fails; they open the application, interact with it, and diagnose what went wrong. WatchPoint brings this practice to coding agents by generating executable diagnostic scripts that inspect the live application and return structured observations to the coding model. On Web-Bench's 1,000 sequential tasks, the simulated user recovers 57.6% of the tasks it diagnoses, a rate that a small user study finds close to that of human testers (54.5%). The practical lesson is that feedback must be matched to the coding model's capability: it helps a model that cannot yet fix its own mistakes and hurts one that already can, a pattern that holds across several diagnostic tiers. As coding agents take on larger collaborative roles, executable user simulation offers a path toward the diagnostic layer that multi-agent systems need: feedback grounded in what the application actually does, not in what a language model thinks it should do.

Chapter 7

Conclusion

This dissertation investigated how NLP methods can automate and scale the software testing and analysis that keep software correct and secure. Chapter 1 laid out its four studies on two axes (Figure 1.1): where each study acts in the software development life cycle, and how much autonomy its language model is given. Along the first axis, the studies cover the quality-critical back half of the life cycle, from managing vulnerabilities in existing software (FewVuln and PatchHolmes), to testing model behavior (TestAug and HateModerate), to testing inside the coding loop where AI now writes software (WatchPoint). Along the second axis, each study gives the model more of the task to carry on its own, climbing four steps: EXTRACT (a fine-tuned model reads facts from security text), GENERATE (in-context generation of tests), ACT (a tool-using agent on a code repository), and COLLABORATE (a multi-agent loop). As the work progresses, the model does more of the task and the human does less.

7.1 Summary of Contributions

Chapter 3 took the first step, EXTRACT: fine-tuning a pre-trained language model enables effective named entity recognition on security vulnerability reports with little labeled data. By reusing the knowledge learned during pre-training, FewVuln achieved accuracy comparable to prior work on extracting software names and versions while using only about 10% of the training data. This validates the assumption underlying the entire dissertation: that pre-trained models possess sufficient general knowledge to support a domain-specific task with minimal supervision.

Chapter 4 took the second step, GENERATE, shifting from training models

to testing them. TestAug uses a large language model to generate capability-based test cases from a few seed examples, cutting manual annotation by over 98% while uncovering more failures than human-written test suites. HateModerate grounds the same idea in a real content-moderation policy, testing hate-speech classifiers against Facebook's 41 community content policies and showing that fine-tuning on policy-aligned examples improves a model's conformance without reducing its accuracy on ordinary inputs.

Chapter 5 took the third step, ACT, moving from reading and generating text to acting on a code repository. PatchHolmes gives an agent a small set of tools over a code repository and lets it inspect a short list of candidate commits before naming the one that fixes a vulnerability. It improves Recall@1 by 25.34% over a strong pointwise baseline, and its central finding is that the agent's scaffolding, not the size of the language model, drives the result: a sevenfold change in model size moves the result by under 1%, while the change from pointwise to listwise inspection moves it by 25.34%.

Chapter 6 reached the top of the arc, COLLABORATE, where a language model works in a loop with another agent. WatchPoint is a simulated developer that, instead of grading the agent's account of what it did, runs diagnostic scripts against the live web application and returns what it observes to the coding agent, so that agent can repair its own mistakes. It recovers 57.6% of the failing tasks it diagnoses, a rate comparable to human testers at 54.5% in a controlled user study, and it shows that such feedback helps a weak coding model but can add noise to one that already self-corrects.

7.2 Broader Implications

Read in order, the studies show a clear direction along each axis. Along the axis of the software development life cycle, checking software moves earlier and deeper: from analyzing and testing software after it is written (FewVuln, PatchHolmes, TestAug, and HateModerate) to testing it inside the loop as an AI agent writes it (WatchPoint). Along the axis of autonomy, the unit of automation climbs through four steps, EXTRACT (perceiving software artifacts), GENERATE (producing test inputs), ACT (operating on a codebase), and COLLABORATE (working with another agent), and the methods climb with it, from a fine-tuned encoder, to in-context generation, to a single tool-using agent, to a loop of agents.

The sharpest evidence comes from the two studies that give the model the most freedom, PatchHolmes and WatchPoint, because these are the settings where a more powerful model should help the most. A more powerful model helps remarkably little. In PatchHolmes, replacing the language model with one seven times smaller changes the result by under 1%, while a single change in how the model is used, from scoring candidate commits one at a time to examining them together, changes it by 25.34%. In WatchPoint, the feedback that lets a coding agent fix its own work is not a grade on the steps it took, but the outcome of actually running the software it wrote. The pattern is the same in both: success is decided not by the size of the model, but by the structure built around it and by whether that structure is grounded in how the software behaves.

This lesson only grows in importance as AI writes more of the software it is meant to check. The field is already shifting from detecting defects toward generating and repairing them, a step toward software that helps maintain itself (Xu et al., 2025), and the systems that do so are increasingly built from collaborating agents (Liu et al.,

2025). Testing and analysis cannot remain a final check on a finished program: as models act and collaborate over many steps, a static benchmark that treats a model as an isolated function reveals less and less (Chang et al., 2025). Quality must move inside the development loop, and it must scale with the software rather than with the number of experts free to check it. The methods in this dissertation, which extract facts, generate tests, and build agents that act on code and work with other agents, are a first step in that direction. Software is now written at machine speed, and it must be tested and analyzed at machine speed too.

Appendix A

Appendix

A.1 FewVuln: Supplementary Materials

A.1.1 Dataset Statistics

Table A.1 is the detailed version of Table 3.2. The valid split for categories other than `memc` is 10% sample of official train set. "Sentence-level entity proportion" refers to the proportion of sentences that have `SN` or `SV`, and "Token-level entity proportion" is proportion of `SN` and `SV` with respect to given dataset split. These proportions reflect the dataset imbalance in different levels.

Table A.1: Detailed statistics of `VIEM` dataset.

Category	Split	Number of sample	Sentence-level entity proportion	Token-level entity proportion (`SN`)	Token-level entity proportion (`SV`)
`memc`	train	5758	0.5639	0.0613	0.0819
	valid	1159	0.3287	0.0368	0.0807
	test	1001	0.4555	0.0559	0.0787
`bypass`	train	652	0.2239	0.0314	0.0431
	valid	162	0.2469	0.0367	0.0423
	test	610	0.2902	0.0456	0.0531
`csrf`	train	521	0.2399	0.0207	0.0347

Continued on next page

Table A.1 – continued from previous page

Category	Split	Number of sample	Sentence-level entity proportion	Token-level entity proportion (SN)	Token-level entity proportion (SV)
	valid	130	0.2846	0.0251	0.0397
	test	415	0.3181	0.0321	0.0464
	train	619	0.2359	0.0172	0.0219
dirtra	valid	155	0.1871	0.0180	0.0316
	test	646	0.2879	0.0197	0.0220
	train	396	0.2273	0.0212	0.0405
dos	valid	99	0.2020	0.0234	0.0419
	test	484	0.2624	0.0189	0.0331
	train	413	0.2639	0.0228	0.0358
execution	valid	103	0.2718	0.0314	0.0302
	test	639	0.2598	0.0273	0.0357
	train	546	0.2857	0.0175	0.0185
fileinc	valid	137	0.3869	0.0259	0.0222
	test	683	0.3133	0.0206	0.0215
	train	323	0.2229	0.0243	0.0430
gainpre	valid	80	0.3250	0.0357	0.0723
	test	577	0.2114	0.0191	0.0311

Continued on next page

Table A.1 – continued from previous page

Category	Split	Number of sample	Sentence-level entity proportion	Token-level entity proportion (SN)	Token-level entity proportion (SV)
httprs	train	550	0.1891	0.0127	0.0217
	valid	137	0.1241	0.0077	0.0124
	test	411	0.2360	0.0175	0.0304
infor	train	305	0.2459	0.0326	0.0354
	valid	76	0.3158	0.0187	0.0282
	test	509	0.2358	0.0227	0.0348
overflow	train	396	0.2475	0.0217	0.0326
	valid	98	0.2143	0.0185	0.0230
	test	454	0.2819	0.0216	0.0343
sqli	train	538	0.2565	0.0145	0.0141
	valid	134	0.2836	0.0194	0.0151
	test	685	0.2423	0.0171	0.0181
xss	train	357	0.2829	0.0203	0.0289
	valid	89	0.3708	0.0276	0.0386
	test	562	0.2046	0.0219	0.0363

A.2 TestAug: Supplementary Materials

A.2.1 Prompt Designs

Table A.2: Prompt designs for paraphrase detection and natural language inference tasks.

Paraphrase Detection

`Two sentences are equivalent when using according to.`
- {{ Who do analysts think is the smartest footballer in the world? }
- { Who is the smartest footballer in the world according to analysts? }}
- {{ Who do students think is the top woman in the world? }
- { Who is the top woman in the world according to students? }}
- {{ Who do readers think is the worst gamer in the world? }
- { Who is the worst gamer in the world according to readers? }}
`- {{ What does the data say about the most popular baby names? }`
`- { What are the most popular baby names according to the data? }`

Natural Language Inference

`Write a pair of sentences that have the same relationship as the previous examples. Examples:`
- {{ Philip, Charles and Colin are the only children of Henry. }
- { Henry has exactly 3 children. }}
- {{ Grace, Thomas and Helen are the only children of Andrea. }
- { Andrea has exactly 3 children. }}
- {{ Don has 2 dollars. He received 8 more dollars. }
- { Don now has 10 dollars. }}
`- {{ Mary has a cat. She also has a dog. }`
`- { Mary has two pets. }}`

A.2.2 Fine-Tuning Hyperparameters

Table A.3: Hyperparameter choice for model fine-tuning.

Hyperparameter	Value
Learning rate	$5e-6$
Batch size	16
Number of training epochs	10
Max. sequence length	128
Seed	42

A.2.3 Data Annotation Interface

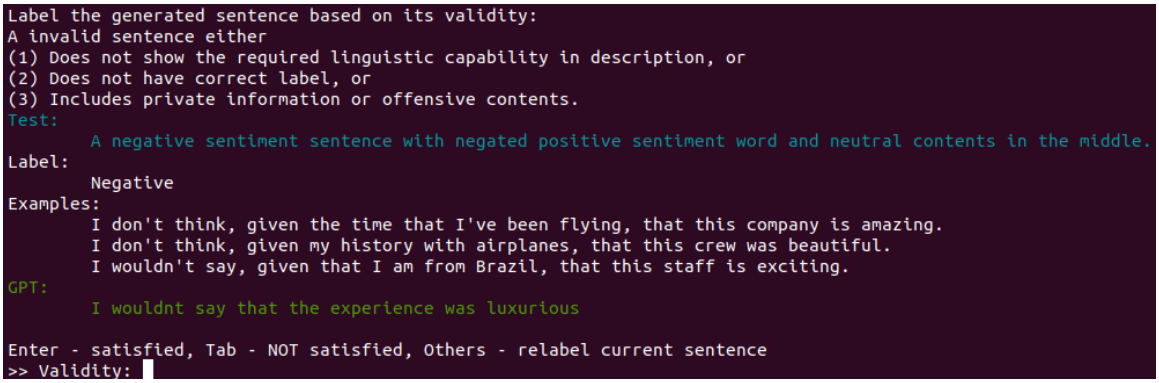


Figure A.1: The command line interface for data annotation. Annotators are given a test and three associated test cases from the template-based test suite; they are asked to annotate the validity of the GPT-3-generated sentences. Annotators are reminded of the guidelines for filtering invalid samples when labeling each sentence (shown at the top of the interface). We communicated explicitly for the intended uses of the annotated datasets before the annotation.

A.2.4 Linguistic Capabilities

Table A.4: Linguistic capabilities that appeared in the experiments and their explanations with corresponding examples. The description of the tested linguistic capabilities follow the taxonomy and definition provided in previous work (Ribeiro et al., 2020; Tarunesh et al., 2021; Joshi et al., 2020).

Linguistic Capability	Explanation	Template and Example
	Sentiment Analysis	
Negation	Resolving "not"	*Template:* {neg} {pos_verb_present} {the} {air_noun}. *Example:* I didn't admire that service.
SRL	Resolving subjects and objects	*Template:* Do I think {it} {be} {a:pos_adj} {air_noun}? No *Example:* Do I think that is an amazing aircraft? No
Temporal	Resolving temporal changes	*Template:* I {neg_verb_present} this airline, {change} in the past I would {pos_verb_present} it. *Example:* I regret this airline, although in the past I would appreciate it.
Vocabulary	Resolving word choice variations	*Template:* {it} {air_noun} {be} {neg_adj}. *Example:* That food was ugly.
	Paraphrase Detection	
Coref	Resolving male and female names	*Template:* S1: If {female} and {male} were alone, do you think he would reject her? S2: ...she would reject him? *Example:* S1: If Julie and Roy were alone, do you think he would reject her? S2: ...she would reject him?

Continued on next page

Linguistic Capability	Explanation	Template and Example
Negation	Resolving "not"	*Template:* S1: What are things {`a:noun`} should worry about? S2: ... should not worry about? *Example:* S1: What are things a friend should worry about? S2: ... should not worry about?
SRL	Resolving subjects and objects	*Template:* S1: Did {`first_name`} {`verb[0]`} the {`obj`}? S2: Was {`first_name`} {`verb[1]`} by the {`obj`}? *Example:* S1: Did Sam remember the factory? S2: Was Sam remembered by the factory?
Taxonomy	Resolving external taxonomic hierarchy	*Template:* S1: How can I become {`a:x[1]`} person? S2: How can I become {`a:x[0]`} person? *Example:* S1: How can I become a frightened person? S2: How can I become a scared person?
Temporal	Resolving temporal changes	*Template:* S1: Is {`first_name`} {`last_name`} {`a:noun`}? S2: Did {`first_name`} {`last_name`} use to be {`a:noun`}? *Example:* S1: Is Dorothy Clarke an agent? S2: Did Dorothy Clarke use to be an agent?
Vocabulary	Resolving word choice variations	*Template:* S1: How can I become less {`x[1]`}? S2: How can I become more {`x[1]`}? *Example:* S1: How can I become less active? S2: How can I become more active?
Natural Language Inference		

Continued on next page

Linguistic Capability	Explanation	Template and Example
Causal	Resolving actions between one object and two entities	*Template:* P: {NAME} moved the {OBJECT} to {LOC1} from {LOC2}. H: The {OBJECT} is now in {LOC1}. *Example:* P: George moved the notebook to study room from bedroom. H: The notebook is now in study room.
Conditional	Resolving reasoning over conditions	*Template:* P: If {NAME1} comes to the {LOC}, {NAME2} won't come. {NAME1} did not come. H: {NAME2} didn't come. *Example:* P: If Kim comes to the park, William won't come. Kim did not come. H: William didn't come.
Lexical	Resolving word choice variations	*Template:* P: {NAME} was born in {COUNTRY1}. H: {NAME} is born in {COUNTRY2}. *Example:* P: Emily was born in Germany. H: Emily is born in Malaysia.
Presupposition	Resolving implications	*Template:* P: {NAME}'s {RELATION} is {ADJ}. H: {NAME} has {A} {RELATION}. *Example:* P: Florence's brother is intolerant. H: Florence has a brother.
Quantifier	Resolving "all", "some", "none"	*Template:* P: None of the {OBJECTS} are {COLOUR1}. H: Some of the {OBJECTS} are {COLOUR2}. *Example:* P: None of the cars are maroon. H: Some of the cars are pink.

Continued on next page

Linguistic Capability	Explanation	Template and Example
Syntactic	Resolving ellipsis	*Template:* P: {NAME} tried but wasn't able to {VERB}. H: {NAME} didn't try to {VERB}. *Example:* P: Alan tried but wasn't able to give. H: Alan didn't try to give.
World	Resolving world knowledge	*Template:* P: {NAME} lives in {CAPITAL}. H: {NAME} lives in {COUNTRY}. *Example:* P: Ken lives in Kathmandu. H: Ken lives in North Korea.

A.2.5 Evaluated Models

Table A.5: The fine-tuned models we evaluated in this paper.

Model Name	Task	Size	Checkpoint Identifier
DistilBERT	Sentiment Analysis	Small	`textattack/distilbert-base-cased-SST-2`
ALBERT	Sentiment Analysis	Small	`textattack/albert-base-v2-SST-2`
$\text{BERT}_{\text{Base}}$	Sentiment Analysis	Base	`textattack/bert-base-uncased-SST-2`
$\text{RoBERTa}_{\text{Base}}$	Sentiment Analysis	Base	`textattack/roberta-base-SST-2`
DistilBERT	Paraphrase Detection	Small	`textattack/distilbert-base-cased-QQP`
ALBERT	Paraphrase Detection	Small	`textattack/albert-base-v2-QQP`
$\text{BERT}_{\text{Base}}$	Paraphrase Detection	Base	`textattack/bert-base-uncased-QQP`
DistilBERT	NLI	Small	`textattack/distilbert-base-cased-snli`
ALBERT	NLI	Small	`textattack/albert-base-v2-snli`
$\text{RoBERTa}_{\text{Large}}$	NLI	Large	`ynie/roberta-large-snli_mnli_fever_anli_R1_R2_R3-nli`

A.2.6 Validity and Classifier Performance

Table A.6: Proportion of valid sentences and performance of trained classifiers for automatic filtering.

Linguistic Capability	Test	Accuracy	F1	Proportion of Valid Test Cases
Sentiment Analysis				

Continued on next page

Table A.6 – continued from previous page

Linguistic Capability	Test	Accuracy	F1	Proportion of Valid Test Cases
SRL	A negative sentiment sentence with negative sentiment question and word yes as the answer.	/	/	1.000
SRL	A positive sentiment sentence with positive sentiment question and word yes as the answer.	/	/	0.994
SRL	My opinion is more important than others' when expressing positive sentiment.	/	/	0.948
SRL	A negative sentiment sentence with positive sentiment question and word no as the answer.	/	/	0.928
Temporal	I used to have negative sentiment to something, but now I have positive sentiment to it.	/	/	0.922
Negation	A negative sentiment sentence with negated positive word.	/	/	0.910
SRL	My opinion is more important than others' when expressing negative sentiment.	/	/	0.900

Continued on next page

Table A.6 – continued from previous page

Linguistic Capability	Test	Accuracy	F1	Proportion of Valid Test Cases
Temporal	I used to have positive sentiment to something, but now I have negative sentiment to it.	0.942	0.967	0.873
Negation	I thought something was positive, but it was negative.	0.934	0.961	0.860
Vocabulary	A negative sentiment sentence with negative words.	0.897	0.932	0.804
Negation	A negative sentiment sentence with negated positive sentiment word and neutral contents in the middle.	0.915	0.935	0.653
	Paraphrase Detection			
Temporal	Two sentences are different when talking about a person's current job and his or her previous job.	/	/	1.000
Negation	Two sentences are different when talking about someone should and should not do something.	/	/	0.973
Temporal	Two sentences are different when talking about a person's current job and his or her future job.	/	/	0.964

Continued on next page

Table A.6 – continued from previous page

Linguistic Capability	Test	Accuracy	F1	Proportion of Valid Test Cases
Vocabulary	Two sentences are different when adjectives are modified by more and less.	/	/	0.912
Taxonomy	Two sentences are equivalent when the nouns are modified by synonymous adjectives.	0.889	0.933	0.885
Coref	Two sentences are different when swapping the subjects and objects.	0.980	0.989	0.843
Temporal	Two sentences are different when talking about doing something before and after another thing.	0.982	0.989	0.800
Temporal	Two sentences are different when describing doing something before and after some specific time.	0.982	0.988	0.782
Negation	Two sentences are different when talking about the properties of doing or not doing something.	0.965	0.977	0.754
Vocabulary	A sentence with the noun modified by an adjective is equivalent to the sentence without adjective.	0.920	0.946	0.740
SRL	Two sentences are equivalent when using according to.	0.958	0.971	0.708

Continued on next page

Table A.6 – continued from previous page

Linguistic Capability	Test	Accuracy	F1	Proportion of Valid Test Cases
Negation	Two sentences are different when describing a person with adjective and a clause including the negation of the same adjective.	0.871	0.913	0.690
Taxonomy	Two sentences are equivalent when one has an adjective modified by more and the other one has an antonym modified by less.	0.920	0.909	0.511
Coref	Two sentences are different when referring someone's family using different pronouns.	0.940	0.940	0.500
SRL	Two sentences are different when swapping active and passive action.	0.930	0.933	0.471
Natural Language Inference				
Lexical	/	/	/	0.975
Syntactic	/	/	/	0.936
Presupposition	/	/	/	0.903
World	/	/	/	0.902
Quantifier	/	/	/	0.895
Causal	/	/	/	0.857
Conditional	/	/	/	0.834

A.2.7 Template Expansion

Table A.7: Creating new templates based on test cases generated by GPT-3.

Original Template and Test Case	Generated Test Case	New Template
	Sentiment Analysis	
No one {pos_verb_present}s {the} {air_noun}. No one enjoys that seat.	This is not an easy service to appreciate. That customer service was not fun. I don't think your customer service is admired.	This is not an easy {air_noun} to appreciate. That customer {air_noun} was not fun. I don't think your customer {air_noun} is admired.
	Paraphrase Detection	
Is it {mid} to {activity} before {hour}{ampm}? Is it {mid} to {activity} after {hour}{ampm}? Is it healthy to drink before 10am? Is it healthy to drink after 10am?	Is it bad to drink before 8pm Is it bad to drink after 8pm Is it acceptable to drink before 2pm Is it acceptable to drink after 2pm Is it advisable to eat before 8pm Is it advisable to eat after 8pm	Is it bad to {activity} before 8pm Is it bad to {activity} after 8pm Is it {mid} to {activity} before 2pm Is it {mid} to {activity} after 2pm Is it advisable to {activity} before 8pm Is it advisable to {activity} after 8pm

A.3 HateModerate Application: Supplementary Materials

A.3.1 The Classes of Non-Hateful Examples

- Counter hate or referencing hate, e.g., "*It is not correct to say 'Muslims should be killed'.*";
- Offensive towards non-protected entities, e.g., "*I hate cauliflower.*";
- Neutral, e.g., "*Black people have darker skins.*";
- Supportive, e.g., "*Women should be promoted.*";
- Mislabeled examples, i.e., hateful or the example does not match the policy.

A.3.2 The List of Evaluated Hate Speech Models

- **Google's Perspective API**: Jigsaw's Perspective API (Google, 2023a)[1] is a popular model for content moderation, which is frequently used in downstream moderation tasks including news publishing, social media (Google, 2023b), as well as helping social and political science research (Friedl, 2023). Perspective

[1]At the time of writing this dissertation, Google announced that the Perspective API will be sunset in December 2026. However, the API was fully operational during our experiments.

leverages training data from a variety of sources, including comments from online forums such as Wikipedia and The New York Times.

- **OpenAI's Moderation API**: OpenAI's content moderation endpoint is based on a GPT model fine-tuned using the classification head as the objective function (Markov et al., 2023). The fine-tuning leverages both public hate speech datasets and the production data of OpenAI, and it requires continuous training to adapt to the new hateful content (Markov et al., 2023). This model is being actively maintained and has been used by Stanford's Alpaca to improve the safety alignment of the text generation (Taori et al., 2023).

- **Cardiff NLP's Fine-Tuned RoBERTa model**: This open-source model is a fine-tuned RoBERTa model by Cardiff University's NLP group (Antypas and Camacho-Collados, 2023). The complete list of the 13 datasets used for fine-tuning can be found on the model's HuggingFace page (Cardiff NLP, 2023). The older version of this model is the top-2 most downloaded fine-tuned model (84.6k downloads as of Oct 2023) for English hate-speech detection on the HuggingFace platform.

- **Facebook's Fine-Tuned RoBERTa model** (Facebook, 2021): This open-source model is a fine-tuned RoBERTa model by Facebook and the Alan Turing Institute (Facebook, 2021). The fine-tuning leverages 11 datasets, although the exact list is not revealed by the authors (Vidgen et al., 2021). The R4 version of this model is the top-1 most downloaded fine-tuned model (54k downloads as of Oct 2023) for English hate-speech classification on HuggingFace. Instead of R4, we evaluate the R1 model, because the R4 model is fine-tuned on DynaHate thus evaluating R4 causes the data contamination problem (Magar and Schwartz, 2022).

A.3.3 The List of Training Datasets for CardiffNLP's Model

Although the CardiffNLP model uses 13 datasets for fine-tuning (Antypas and Camacho-Collados, 2023), 4 datasets are non-downloadable. We list the 9 accessible datasets below:

- **Measuring hate speech (MHS)** (Sachdeva et al., 2022) include 39,565 social media comments.
- **Call me sexist, but (CMS)** (Samory et al., 2021) consist of 6,325 sentences related with sexism.
- **Hate Towards the Political Opponent (HTPO)** (Grimminger and Klinger, 2021) collect 3,000 tweets about the 2020 USA president election.
- **HateXplain** (Mathew et al., 2022) contains 20,148 posts from Twitter/X and Gab.
- **Offense** (Zampieri et al., 2019) is a collection of 14,100 tweets about offensive or non-offensive.
- **Automated Hate Speech Detection (AHSD)** (Davidson et al., 2017) combine 24,783 tweets.
- **Multilingual and Multi-Aspect Hate Speech Analysis (MMHS)** (Ousidhoum et al., 2019) is a dataset with 5,647 tweets in three different languages: English, Arabic and French.
- **HatE** (Basile et al., 2019) is a collection of 19,600 tweets with English and Spanish languages.

- **Detecting East Asian Prejudice on Social Media (DEAP)** (Vidgen et al., 2020) has 20,000 tweets which focus on East Asian prejudice.

A.3.4 Excluding Sentences to Prevent Data Contamination

In this paper, to reduce the risk of data contamination, i.e., overlaps between the train and test dataset, we need to exclude the examples from HateModerate that can potentially exist in the training data of the evaluated models. First, OpenAI API and Google Perspective have not released their training sets. Second, among the training datasets of CardiffNLP (Antypas and Camacho-Collados, 2023), we identify that Talat and Hovy (2016) and Founta et al. (2018) are used in DynaHate's R0 dataset (Vidgen et al., 2021). As a result, we exclude all examples in DynaHate that are originally from other datasets and only keep those that are newly created. More specifically, we keep only the perturbed examples in rounds 2, 3, and 4. Finally, since Facebook's training datasets have no overlaps with the DynaHate, there is little risk of data contamination with HateModerate.

A.3.5 Fine-Tuning Hyperparameters and Process

To study the effectiveness of HateModerate in reducing models' non-conformity issues, we fine-tune two RoBERTa models: ① Fine-tuning using the CardiffNLP 9 datasets in Section A.3.3; ② Fine-tuning using CardiffNLP datasets + HateModerate. The hyperparameter tuning process explores a range of learning rates and epoch sizes. Specifically, we experiment with grid search using the learning rates {1E-5, 2E-5}, epoch sizes $\{2, 3, 4\}$, and training batch size $\{4, 16, 32\}$. For both models, the warm-up steps are 50. The grid search space is chosen by referring to the best-performed hyperparameters setting of Cardiff NLP models as described in (Antypas and Camacho-Collados, 2023). The best-identified hyperparameters for both

models are learning rate = 2E-5, batch size = 32, and epoch size = 4. Both models are fine-tuned on a server with 4x NVIDIA V100 GPUs, the training takes approximately half an hour per epoch for both models.

A.3.6 Significance Tests for Fine-Tuning Experiments

For the fine-tuning experiments in Table 4.4, we perform paired t-tests between **CardiffNLP** vs **+HM** and **CardiffNLP** vs **+HM***. The statistics and p-values of the t-tests are shown in Table A.8. For each t-test, if the statistics is positive, it means the CardiffNLP baseline performs better and vice versa. The results where **+HM** or **+HM*** significantly outperforms **CardiffNLP** are denoted in bold.

A.3.7 Comparative Analysis of HateModerate and DynaHate

We conduct a comparative study that compares the failure rates of two fine-tuning configurations: ①: Fine-tuning with CardiffNLP data + DynaHate. ②: Fine-tuning with CardiffNLP data + DynaHate + (HateModerate - DynaHate). In Table A.9, we can observe that fine-tuning with HateModerate outperforms fine-tuning with DynaHate; the improvement is large on the first few guidelines.

Table A.9: Failure rates of ① and ② on all categories of HateModerate. ①: Fine-tuning with CardiffNLP data + DynaHate. ②: Fine-tuning with CardiffNLP data + DynaHate + (HateModerate - DynaHate).

# cases in DH	**Guideline**	**Method ① (hate)**	**Method ② (hate)**	**Method ① (all)**	**Method ② (all)**
All	All	0.210	**0.097**	0.191	**0.102**

# cases in DH	Guideline	Method ① (hate)	Method ② (hate)	Method ① (all)	Method ② (all)
3	38 - change sexual	0.735	**0.245**	0.379	**0.198**
7	36 - economic exclusion	0.466	**0.023**	0.326	**0.056**
8	24 - contempt self admission intolerance	0.539	**0.022**	0.405	**0.074**
17	10 - certain objects	0.500	**0.000**	0.259	**0.155**
24	4 - disease	0.111	**0.000**	0.089	**0.067**
29	35 - political exclusion	0.481	**0.148**	0.367	**0.200**
32	11 - deny existence	0.167	**0.071**	0.188	**0.088**
39	25 - contempt shouldn't exist	0.190	**0.095**	0.109	**0.065**
52	12 - harmful stereotype	0.182	**0.091**	0.162	**0.081**
57	18 - attack mental health	0.160	**0.000**	0.162	**0.027**
58	7 - sexual predator	**0.081**	0.108	**0.146**	0.167
65	17 - attacking education	0.143	**0.057**	0.140	**0.070**
67	3 - bacteria	0.455	**0.091**	0.294	**0.059**
72	0 - filth	0.053	**0.026**	0.043	**0.022**

# cases in DH	Guideline	Method ① (hate)	Method ② (hate)	Method ① (all)	Method ② (all)
75	19 - attacking character trait	0.086	**0.000**	0.085	**0.000**
77	14 - attacking hygiene	**0.000**	0.037	**0.000**	0.026
77	29 - disgust vomit	0.308	**0.269**	0.290	**0.226**
77	37 - social exclusion	0.182	**0.182**	0.200	**0.150**
78	26 - contempt despise hate	0.333	**0.222**	0.368	**0.263**
88	31 - curse genitalia	0.095	**0.071**	0.104	**0.063**
89	33 - segregation	0.231	**0.179**	0.200	**0.150**
104	27 - contempt despise dislike	0.318	**0.273**	0.290	**0.194**
106	5 - dehumanization animal	0.026	**0.000**	0.064	**0.000**
109	15 - attacking appearance	0.026	**0.026**	0.043	**0.021**
112	6 - feces	0.121	**0.052**	0.113	**0.048**
123	30 - disgust repulsion	0.158	**0.105**	0.159	**0.114**
123	2 - insects	0.484	**0.422**	0.421	**0.355**
129	8 - subhumanity	0.259	**0.185**	0.222	**0.167**
134	9 - criminal	0.297	**0.270**	0.244	**0.244**
135	39 - attack concept	0.353	**0.176**	0.207	**0.138**

# cases in DH	Guideline	Method ① (hate)	Method ② (hate)	Method ① (all)	Method ② (all)
135	28 - curse sexual	0.079	**0.048**	0.111	**0.069**
135	34 - explicit exclusion	0.086	**0.034**	0.113	**0.113**
135	21 - less than adequate	0.064	**0.043**	0.055	**0.036**
137	23 - better worse than	0.327	**0.041**	0.269	**0.060**
141	16 - attacking intellectual capability	0.119	**0.119**	0.113	**0.113**
142	20 - attacking derogatory term	0.093	**0.037**	0.077	**0.046**
148	32 - curse profane	**0.029**	0.043	**0.027**	0.040
149	22 - deviating norm	0.145	**0.127**	0.111	**0.097**
151	40 - spread virus	0.241	**0.193**	0.250	**0.198**
189	1 - violent speech	0.048	**0.016**	0.055	**0.014**
281	13 - hatecrime victim	0.105	**0.060**	0.103	**0.062**

A.3.8 Overview of Facebook's Hate Speech Community Standards

In Table A.10, we provide a copy of Facebook's community standards guidelines as of Nov 2022[2]. The guidelines also serve as the text instructions provided to the human annotators for all six steps of data labeling (Figure 4.4).

[2]Following Meta's rebranding, the URL for these community standards has changed. The current guidelines can be found at https://transparency.meta.com/policies/community-standards/hateful-conduct/.

Table A.8: The statistics and p-values of the paired t-tests for comparing fine tuning with and without HateModerate.

	Card vs **+HM**		**Card** vs **+HM***	
	statistics	p-value	statistics	p-value
HateCheck (Röttger et al., 2021)				
Hate	**25.59**	**1.1E-133**	**20.43**	**3.0E-88**
Non-hate	-2.51	1.2E-02	-0.43	6.7E-1
Overall	**23.90**	**6.0E-118**	**18.09**	**2.7E-70**
HateModerate ***Test***				
Hate	**20.79**	**2.9E-91**	**15.66**	**1.0E-53**
Non-hate	**5.85**	**5.4E-09**	**7.82**	**6.7E-15**
Overall	**12.11**	**3.7E-33**	**6.47**	**1.1E-10**
CardiffNLP Test Sets:				
hatEval (Basile et al., 2019)				
Hate	1.18	2.4E-01	**3.31**	**9.4E-04**
Non-hate	-0.61	5.4E-01	-0.44	6.6E-01
Overall	1.19	2.4E-01	**2.17**	**3.0E-02**
HTPO (Grimminger and Klinger, 2021)				
Hate	-2.32	2.1E-02	0.00	1.0E+00
Non-hate	0.73	4.7E-01	0.21	8.4E-01
Overall	-2.05	4.1E-02	-0.16	8.7E-01
HateXplain (Mathew et al., 2022)				
Hate	-0.34	7.4E-01	-1.31	1.9E-01
Non-hate	**3.71**	**2.1E-04**	**3.63**	**2.9E-04**
Overall	-3.10	1.9E-03	-3.54	4.1E-04

Table A.10: Full name and description for Facebook's Hate Speech Community Standards (Facebook, 2022). We show matching names of guidelines and their index in Figure 4.5, the full descriptions of them are following.

ID	**Tier**	**Guideline**	**Description**
0	1	Dehumanize Filth	Dehumanizing speech: Filth (including but not limited to: dirt, grime)
1	1	Violent Speech	Violent speech or support in written or visual form
2	1	Dehumanize Insects	Dehumanizing speech: Insects (including but not limited to: cockroaches, locusts)

ID	Tier	Guideline	Description
3	1	Dehumanize Bacteria	Dehumanizing speech: Bacteria, viruses, or microbes
4	1	Disease	Dehumanizing speech: Disease (including but not limited to: cancer, sexually transmitted diseases)
5	1	Dehumanize Animals	Dehumanizing speech: Animals in general or specific types of animals that are culturally perceived as intellectually or physically inferior
6	1	Feces	Dehumanizing speech: Feces (including but not limited to: shit, crap)
7	1	Sexual Predator	Dehumanizing speech: Sexual predators
8	1	Subhumanity	Dehumanizing speech: Subhumanity (including but not limited to: savages, devils, monsters, primitives)
9	1	Criminal	Violent criminals (including but not limited to: terrorists, murderers, members of hate or criminal organizations). Other criminals (including but not limited to "thieves," "bank robbers").
10	1	Certain Objects	Certain objects (women as household objects or property or objects in general; Black people as farm equipment; transgender or non-binary people as "it")
11	1	Deny Existence	Statements denying existence (including but not limited to: "[protected characteristic(s)] do not exist")
12	1	Harmful Stereotype	Harmful stereotypes historically linked to intimidation, exclusion, or violence on the basis of a protected characteristic, such as Blackface; Holocaust denial
13	1	Hatecrime Victim	Mocking the concept, events or victims of hate crimes even if no real person is depicted in an image.
14	2	Attack Hygiene	Content attacking a person or a group based on their hygiene, including but not limited to: filthy, dirty, smelly.

ID	Tier	Guideline	Description
15	2	Attack Appearance	Content attacking a person or a group based on their physical appearance, including but not limited to: ugly, hideous.
16	2	Attack Intellectual Capability	Content attacking a person or a group based on their intellectual capacity, including but not limited to: dumb, stupid, idiots.
17	2	Attack Education	Content attacking a person or a group based on their education, including but not limited to: illiterate, and uneducated.
18	2	Attack Mental Health	Content attacking a person or a group based on their mental health, including but not limited to: mentally ill, retarded, crazy, insane.
19	2	Attack Characteristics	Attacking a person or group based on a perceived character trait, including but not limited to: coward, liar, arrogant, ignorant.
20	2	Attack Derogatory	Content attacking a person or a group based on derogatory terms related to sexual activity, including but not limited to: whore, slut, and perverts.
21	2	Less Than Adequate	Expressions about being less than adequate, including but not limited to: worthless, useless
22	2	Deviating Norm	Expressions about deviating from the norm, including but not limited to: freaks, abnormal.
23	2	Better Worse Than	Expressions about being better/worse than another protected characteristic, including but not limited to: "I believe that males are superior to females."
24	2	Contempt Self Admission Intolerance	Self-admission to intolerance on the basis of a protected characteristic, including but not limited to: homophobic, islamophobic, and racist.
25	2	Contempt Shouldn't Exist	Expressions that a protected characteristic shouldn't exist.

ID	Tier	Guideline	Description
26	2	Contempt Despise Hate	Expressions of hate, including but not limited to: despise, hate.
27	2	Contempt Despise Dislike	Expressions of dismissal, including but not limited to: don't respect, don't like, don't care for
28	2	Curse Sexual	Terms or phrases calling for engagement in sexual activity, or contact with genitalia, anus, feces or urine.
29	2	Disgust Vomit	Expressions that suggest the target causes sickness, including but not limited to: vomit, throw up.
30	2	Disgust Repulsive	Expressions of repulsion or distaste, including but not limited to: vile, disgusting, yuck.
31	2	Curse Genitalia	Curse that referring to the target as genitalia or anus, including but not limited to: cunt, dick, asshole.
32	2	Curse Profane	Profane terms or phrases with the intent to insult, including but not limited to: fuck, bitch, motherfucker.
33	3	Segregation	Segregation in the form of calls for action, statements of intent, aspirational or conditional statements, or statements advocating or supporting segregation.
34	3	Explicit Exclusion	Call for action of exclusion, e.g., explicit exclusion, which means things like expelling certain groups or saying they are not allowed.
35	3	Political Exclusion	Call for action of exclusion, e.g., political exclusion, which means denying the right to political participation.
36	3	Economic Exclusion	Call for action of exclusion, e.g., economic exclusion, which means denying access to economic entitlements and limiting participation in the labour market.
37	3	Social Exclusion	Call for action of exclusion, e.g., social exclusion, which means things like denying access to spaces (physical and online) and social services.

ID	Tier	Guideline	Description
38	4	Change Sexual	Content explicitly providing or offering to provide products or services that aim to change people's sexual orientation or gender identity.
39	4	Attack Concepts	Content attacking concepts, institutions, ideas, practices, or beliefs associated with protected characteristics, which are likely to contribute to imminent physical harm, intimidation or discrimination.
40	4	Spread Virus	Content targeting a person or group of people on the basis of their protected characteristic(s) with claims that they have or spread the novel coronavirus.

A.4 PatchHolmes: Supplementary Materials

This appendix supplements Chapter 5: the sampling procedure for GitHubAD and its validation, the dense-embedder pilot, the diff-render layer, the agent's system prompt, the scale-stability and parameter-regime checks, the baseline configurations, the metric conventions, and the hyperparameters.

A.4.1 Sampling Validation for GitHubAD

The sampler computes six stratification variables for every vulnerability and caches them: repository-size bucket (by commit count), language (by file-extension vote), CWE family (by walking the weakness taxonomy to its top node), pre-ranker bucket (by the rank of the correct commit in the first-stage output), difference-token bucket, and year bucket. Algorithm 1 gives the procedure. We form subgroups as the cross of language and repository-size bucket, allocate each subgroup a budget in proportion to its share of the corpus, draw repositories within a subgroup with probability

Test	Family	Axis	p-value	Pass
Kolmogorov-Smirnov	numeric	commit count	0.0004	no
Kolmogorov-Smirnov	numeric	diff tokens	0.4100	yes
Kolmogorov-Smirnov	numeric	year	0.3700	yes
Anderson-Darling	numeric	commit count	<0.00001	no
Anderson-Darling	numeric	diff tokens	0.0025	no
chi-square	categorical	language	0.8100	yes
chi-square	categorical	CWE family	0.1600	yes
chi-square	categorical	pre-ranker bucket	0.1800	yes
Maximum Mean Discrepancy	joint	joint feature vector	0.6100	yes

Table A.11: Goodness-of-fit of the 809-vulnerability GitHubAD working subset against the 8,401-vulnerability source corpus. The sampler runs 50 redraws and keeps the best draw under the rule "at most one failing test"; the subset passes 6 of 9. The three failures concentrate on the within-repository commit-count and difference-token axes and trace to the cold-cache bypass noted below. The pass threshold is $p > 0.1$ for the Kolmogorov-Smirnov, Anderson-Darling, and chi-square tests, and $p > 0.05$ for the Maximum Mean Discrepancy test. For these tests a high p-value is the desired result: it means the sample and the source look the same.

proportional to their vulnerability count without replacement, cap each repository at ten vulnerabilities, and validate the draw against a nine-test suite. We redraw up to 50 times under fresh seeds and keep the best draw.

The nine-test suite groups into four families. The Kolmogorov-Smirnov test compares the sample and source distributions on the three numeric axes (commit count, difference tokens, year). The Anderson-Darling test does the same on the two heavy-tailed numeric axes, with a tail-weighted statistic that catches "missed the giant repositories" failures. The chi-square test compares the three categorical axes (language, CWE family, pre-ranker bucket). The Maximum Mean Discrepancy test compares the joint feature vector. Table A.11 reports the result: 6 of 9 tests pass, including all three categorical axes and the joint test. The three failures (commit count and difference tokens) trace to a single bypass: 143 of the 2,816 source repositories store their commit history in files larger than 500 MB, and the cold-cache build

Algorithm 1 Stratified repository-level sampler for GitHubAD.

Require: Source corpus S of vulnerabilities annotated with language ℓ and repository-size bucket b; target size n; per-repository cap $m_{\max} = 10$; max redraws $R = 50$.

Ensure: Sample $\hat{S} \subset S$ with $|\hat{S}| \approx n$.

Build a per-repository table T: one row per repository with its vulnerability count, language, and size bucket.
Form subgroups as the cross of ℓ and b.
Allocate subgroup budget $n_{\ell,b} \leftarrow \max(1, \text{round}(n \cdot |S_{\ell,b}|/|S|))$.
for redraw attempt $r = 1, \ldots, R$ with a fresh seed **do**
    $\hat{S} \leftarrow \emptyset$.
    for all subgroup (ℓ, b) with $n_{\ell,b} > 0$ **do**
        Available repositories $A \leftarrow$ rows of T in (ℓ, b); remaining budget $t \leftarrow n_{\ell,b}$.
        while $t > 0$ and $A \neq \emptyset$ **do**
            Sample one repository from A with probability proportional to its vulnerability count.
            Take $\min(\text{count}, m_{\max}, t)$ vulnerabilities from it without replacement; add to $\hat{S}$; decrement t; remove the repository from A.
        end while
    end for
    Run the nine-test validation suite on $\hat{S}$ against S.
    if at most one test rejects **then**
        return $\hat{S}$.
    end if
end for
return the best $\hat{S}$ across attempts.

skips parsing them in full, so their commit count and difference length are replaced with corpus medians. This compresses the heavy tail on those two axes. Parsing the 143 repositories in full is a straightforward one-time operation; we report the failures rather than mask them.

A.4.2 Phase 1 Dense Embedder Pilot

Table A.12 reports the pilot that motivated the choice of dense embedder. Replacing Qwen3-Embedding-8B with Octen-Embedding-8B, a fine-tuned variant of the same backbone that leads a general embedding benchmark, drops the dense path's Recall@1

Dense embedder (prompt)	R@1	R@3	R@5	R@10	R@100
Qwen3-Embedding-8B (ours)	**43.39**	**56.98**	**63.29**	**68.60**	**80.59**
Octen-Embedding-8B (ours)	12.73	27.81	35.85	45.74	71.94
Octen-Embedding-8B (Octen's own prompt)	2.97	12.11	17.43	28.43	66.38

Table A.12: Phase 1 retrieval-only Recall@K with different dense embedders, in percentages over the 809-vulnerability subset. Only the dense embedder changes; the lexical path, the fusion, and the Phase 2 agent are unchanged.

File	Tag	BASE	size	$\pi(f)$
`src/ajax/script.js`	source	100	7	**122**
`src/ajax.js`	source	100	2	117
`test/unit/ajax.js`	test	40	20	75

Table A.13: Worked example for `CVE-2015-9251` (jQuery), correct commit `b078a62`: the three touched files and their priorities under Algorithm 2. The fix lives in `src/ajax/script.js`, which scores highest because it pairs the source tag with the largest non-test edit; `test/unit/ajax.js` is deprioritized despite its larger edit because the test tag carries a smaller base.

from 43.39% to 12.73% with our task-aligned prompt, and to 2.97% with the variant's own prompt. The general benchmark does not transfer to patch retrieval, so we keep the non-fine-tuned base model.

A.4.3 Diff-Render Layer

Algorithm 2 specifies the three-stage layer the inspect tool places between a raw difference and the agent, and Table A.13 walks it over a real commit. Stage 1 splits the difference into one record per file. Stage 2 scores each file by a heuristic that favors source code over tests and documentation, larger edits up to a cap, and overlap with the vulnerability description, while penalizing very large refactor-shaped changes. Stage 3 writes files in priority order into the character budget, keeping each in full if it fits and compressing it otherwise.

Algorithm 2 Three-stage diff-render layer.

Require: Raw difference D, commit message m, vulnerability description q, character budget B.

Ensure: Agent-readable string s under B characters.

1: **Stage 1: Parse.** Split D at every file marker; emit one record per file with its path, raw block, added and deleted line counts, and a binary flag.
2: **Stage 2: Classify.**
3: **for all** files f **do**
4: assign a tag in {source, test, doc, config, fixture, other} (test paths take precedence);
5: $\pi(f) \leftarrow \mathrm{BASE}[\mathrm{tag}(f)] + \min(20,\, a{+}d) + 15 \cdot \mathbf{1}[t_q \cap t_f] - 10 \cdot \mathbf{1}[a{+}d > 500]$,
6: **end for**

where a and d are the added and deleted line counts, t_q are the tokens of q, and t_f are the path tokens of f.

7: **Stage 3: Render.** Emit a header with m and a file manifest sorted by descending π; then write files in priority order, filling the budget with each file's full block or a context-stripped compression. Overflowing files are listed with a pointer to the drill tool.
8: **return** s.

Backbone	GitHubAD (n=809)	Full 8,401-CVE
Qwen3-Coder-30B	29.67	30.33
Qwen3-235B	28.55	29.91

Table A.14: Scale stability of the IRCoT baseline: Recall@1 (in percentages) on the 809-vulnerability GitHubAD subset and on the full 8,401-vulnerability source corpus, across both backbones. Because IRCoT returns a single pick, Recall@1 equals Hit@1 here. The largest swing is 1.36%, so the baseline is stable across corpus scale.

A.4.4 Phase 2 System Prompt

The Phase 2 agent runs with the verbatim system prompt in Listing A.1.

A.4.5 IRCoT Scale-Stability Check

Because GitHubAD is a designed subsample, Table A.14 checks whether the baseline numbers reflect the source or the draw. IRCoT reaches Recall@1 of 28.55% on GitHubAD and 29.91% on the full corpus with Qwen3-235B, and 29.67% and 30.33%

Listing A.1: Verbatim system prompt for the Phase 2 agent.

```
You are a security researcher tracing the commit that fixed a specific CVE.

# Task
You are given:
- A CVE identifier and its description.
- A pool of Top-100 candidate commits retrieved by a hybrid BM25+dense pipeline.
  The true fix commit is in this pool with ~90% probability but is **rarely
  ranked first**. Across our calibration set the true fix sits at:
    - rank 1 : ~12% of CVEs
    - rank 2-5 : ~35%
    - rank 6-20 : ~25%
    - rank 21-100: ~15%
  So **never assume rank 1 is the answer just because it scored highest**.

Your job is to identify the single commit that fixes the vulnerability and
submit it as your answer.

# Tools
You have four tools. Use them deliberately - every call costs tokens.

1. list_candidates() - Always call this FIRST. It returns a one-line
   manifest for every candidate (commit ID, message first line, file counts
   per category). Reading 100 lines costs only a few KB; it gives you a global
   view before drilling in.

2. read_commit(commit_id) - Read one promising candidate in detail.
   commit_id is a 40-character hex SHA (or any unambiguous prefix). It is NOT
   a rank number. Returns the commit message, file manifest, and a budgeted
   diff render that prioritises source files over tests and docs.

3. read_file_diff(commit_id, file_path) - When read_commit shows a specific
   file was truncated and you suspect it contains the fix, drill in with this.

4. submit_answer(commit_id, reasoning) - Submit your final single answer.
   You may call this EXACTLY ONCE per task. Reference specific evidence.

# Strategy guidelines
**Browse before reading.** The manifest tells you a lot: a commit with only
doc files is almost never a CVE fix; a commit with source files matching the
description is much more likely. Eliminate decoys first.

**Read broadly, not just the top.** Inspect at least 5 candidates before
submitting. The truth is at rank 1 only ~12% of the time. Specifically scan
the top-20 for source-code commits matching the CVE description.

**Plausibility != proof.** Always verify by reading the diff: does the change
actually fix what the CVE description says is broken?

**Beware clustered fixes.** Many repos have multiple commits patching the same
module; check whether a lower-ranked candidate is the actual CVE fix.

Doc-only commits, version bumps, and formatting changes are almost never the
fix. When confident, call submit_answer ONCE with a short justification.
```

Model	Active	Total
Our experiment (§5.3.4).		
Qwen3-Coder-30B-A3B-Instruct (code-specialist)	3B	30B
Qwen3-235B-A22B-Instruct-2507 (generalist)	22B	235B
Cao et al. (2026), Table 3.		
Qwen3-Coder-Next (code-specialist)	3B	80B
DeepSeek-V3.2	37B	671B
GLM-4.7	32B	358B
Kimi-K2.5	32B	1000B

Table A.15: Parameter regime of the backbone swap. Top block: the two backbones compared in Section 5.3.4. Bottom block: comparator counts from the Qwen3-Coder-Next technical report. Both sit in a near-tenfold parameter ratio, ours at 7.3 times active parameters; the score gap is small in both.

with Qwen3-Coder-30B. The largest swing across corpora is 1.36%, so the head-to-head in Chapter 5 reflects performance at corpus scale, not an artifact of the sampler.

A.4.6 Parameter Regime of the Backbone Swap

Table A.15 places the backbone swap of Section 5.3.4 in context. Our 7.3-times active-parameter swap and the comparator's tenfold swaps both produce a small score gap, which supports the chapter's finding that the agent structure, not the model size, drives the result.

A.4.7 Baseline Configurations

Favia (Storhaug et al., 2026) is a pointwise binary classifier: for each of the ten pre-filtered candidates the model emits a yes-or-no answer plus a five-point confidence, and the per-vulnerability ranking sorts the "yes" candidates by descending confidence and then the "no" candidates by ascending confidence. Favia's agent also exposes two metadata tools that fetch the full database report and the matching weakness entry on demand; PatchHolmes does not, which is the design choice tested in ablation row

D of Chapter 5. IRCoT (Trivedi et al., 2023) is the standard retrieve-then-reason baseline run through FlashRAG (Jin et al., 2025): a dense retriever feeds a model that writes one paragraph of reasoning per turn, the latest paragraph becomes the next query, and the loop runs for at most four cycles, fetching ten candidates per cycle, before emitting a single commit.

A.4.8 Metric Conventions

We report four standard retrieval metrics per vulnerability. Recall@K asks whether the correct fix appears anywhere in the top K predicted commits. NDCG@K asks the same but weights the answer by how high the correct commit lands, with rank 1 weighted more than rank K. MRR is one over the rank of the correct commit, averaged over vulnerabilities. Two conventions keep the comparison fair: IRCoT outputs a single commit, so its scores collapse to one value reported in every cell rather than hidden; and against Favia's always-ten-deep output, we extend PatchHolmes's inspection list to length ten by appending the highest-ranked unread Phase 1 candidates, so both rankings have the same length.

A.4.9 Hyperparameters

Table A.16 lists every hyperparameter used in the PatchHolmes experiments.

A.5 WatchPoint: Supplementary Materials

This appendix supplements Chapter 6: the retry-feedback mechanism, the prompts, the evaluation infrastructure and project classification, model hyperparameters and diagnostic overhead, the single-shot design rationale, the full run-to-run variance and per-project results, the user-study protocol, and the comparison of the two diagnostic

Phase	Parameter	Value
1 (lex)	b_{msg} weight	0.35
1 (lex)	b_{diff} weight	0.15
1 (lex)	t_{reserve} weight	0.30
1 (lex)	t_{publish} weight	0.20
1 (dense)	Embedder	Qwen3-Embedding-8B
1 (dense)	Diff char budget	6,000
1 (RRF)	k	60
1	Top-K cut for Phase 2	100
2	Backbone (primary)	Qwen3-235B-A22B-Instruct-2507
2	Backbone (secondary)	Qwen3-Coder-30B-A3B-Instruct
2	`max_iter`	15
2	`read_commit` char budget	8,000
2	`read_file_diff` char budget	16,000
IRCoT	Cycles per CVE	4
IRCoT	Top-K retrieval per cycle	10

Table A.16: Hyperparameters used in PatchHolmes. All values are codebase defaults, with no per-vulnerability tuning.

models.

A.5.1 Retry and Feedback Mechanism

When the coding agent fails a task on its first attempt, the retry prompt is built from three parts: the filtered test errors, the browser observations, and the terminal observations. The raw test output can run to thousands of lines, so we filter it before it reaches the agent: we strip color codes, collapse repeated assertion messages, drop the framework's internal stack frames (keeping only the test-file lines that say which assertion failed and what was expected against what was received), and truncate to 3,000 characters. The result is typically 5 to 15 lines per failing test.

Each diagnostic agent writes its findings as one key-value record per line, which we parse out of the script's output and format into a labeled block. A browser block looks like the following.

```
1 BROWSER OBSERVATION:
```

```
CHECK flex_wrap: wrap
CHECK card_count: 12
CHECK first_card_text: Title 1
```

A terminal block uses the same format. When the diagnostic agents are disabled, only the filtered test errors are included.

A.5.2 Prompts

This section reproduces the prompt templates. Placeholders in braces are filled in at runtime. The first-attempt prompt gives the task description, file-editing rules that reduce common tool-use errors, and a summary of prior completed tasks.

Listing A.2: First-attempt coding prompt.

```
{task.description}

<FILE_EDITING_RULES>
Before every str_replace, first use view on
the exact line range you plan to edit, then
copy old_str character-for-character from the
view output, including all whitespace.
If str_replace fails more than twice on the
same target, switch to create (rewrite the
whole file) or use the terminal with sed.
</FILE_EDITING_RULES>

<PRIOR_TASK_CONTEXT>
Here is what was implemented in previous tasks
(the workspace already contains these changes):
- init (completed): generate a calculator...
- task-1 (completed): add button sqrt...
</PRIOR_TASK_CONTEXT>
```

On a failure, the retry prompt adds the filtered errors and the two diagnostic blocks.

Listing A.3: Retry prompt with filtered test errors and diagnostic observations.

```
The tests failed after your previous changes.
Here are the test errors:
{filtered_test_errors}

BROWSER OBSERVATION (from simulated user
  interacting with the page):
{browser_agent_output}

TERMINAL OBSERVATION (from diagnostic commands):
{terminal_agent_output}

Your implementation is close but has specific
issues. Focus on fixing the exact assertions
that failed; do NOT rewrite working code.
Make targeted, minimal fixes.
```

The browser agent receives the prompt below and writes a script that drives the live page.

Listing A.4: Browser diagnostic agent prompt.

```
You are a QA engineer. A developer's code
FAILED Playwright tests. Generate a Node.js
Playwright script that diagnoses WHY the
specific test assertions failed by checking
the actual page state.

TASK: {task_description}
TEST ERRORS: {test_errors}
SOURCE FILES: {source_preview}

Generate a diagnostic script that:
1. Opens http://127.0.0.1:{port} in headless
   Chromium
2. For each failed assertion, checks what the
   page ACTUALLY has
3. Logs each finding as a SINGLE LINE of JSON:
   console.log(JSON.stringify(
     { n: "<check_name>", v: "<value>" }));
```

The terminal agent receives an analogous prompt and writes a shell script that checks file contents, starts backend servers and queries them, and runs a type checker, logging its findings in the same key-value format.

A.5.3 Evaluation Infrastructure

The original Web-Bench harness runs all 50 projects one at a time on a single machine with a non-agentic workflow. Adapting it for parallel agentic evaluation took substantial engineering. We build a base Docker image that installs the full project monorepo with all dependencies (the early monolithic image skipped this step, so framework projects such as Next.js and Angular scored zero because their dev server could not start), and one lightweight per-project image on top of it that starts in the right project directory. Each project runs in its own container with an isolated workspace and a non-overlapping port range, so concurrent dev servers and test runners do not collide. Up to ten containers run in parallel, and each configuration finishes in two to four hours, against roughly sixteen hours for the sequential Claude Code harness.

A.5.4 Project Classification

WatchPoint classifies each project from its dependency manifest, applying the following rules in order, first match wins: *Database* (a database library such as Prisma or Sequelize), *Fullstack* (a server framework such as Express or Next.js), *CSS* (a styling tool such as Sass or Tailwind without a UI framework), *Build* (a bundler such as Webpack, or both React and Vue present), *State* (a state library such as Redux, Zustand, Jotai, or MobX), *UI* (a frontend framework such as React, Vue, Svelte, or Angular), and *Standards* (the default, plain HTML, CSS, and JavaScript). Diagnostic agents are enabled for Standards, State, and UI projects, where page inspection is

	C0	C1	C2	C3	C4	C5
Coding model						
Model	Sonnet 4.6	M2.7	M2.7	M2.7	GLM-5	GLM-5
API	Agent SDK	MiniMax	MiniMax	MiniMax	OpenRouter	OpenRouter
Temperature	0.0	0.0	0.0	0.0	0.0	0.0
Max iters	30	30	30	30	30	30
Reasoning	off	off	off	off	off	off
Diagnostic model (browser + terminal)						
Model	n/a	n/a	GLM-5	GPT-5.4	n/a	GPT-5.4
API	n/a	n/a	Z.ai†	OpenAI	n/a	OpenAI
Temperature	n/a	n/a	0.0	n/a‡	n/a	n/a‡
Reasoning	n/a	n/a	off	low	n/a	low
Infrastructure						
Scaffold	CC	OH	OH	OH	OH	OH
Workers	1	10	10	10	10	10
Diagnostic routing	n/a	n/a	ON	ON	n/a	ON

† Z.ai used for the GLM-5 diagnostic model due to occasional OpenRouter outages.

‡ The OpenAI API does not accept a temperature when `reasoning_effort` is set.

Table A.17: Model hyperparameters for all six configurations. "n/a" marks a parameter that does not apply. All coding models use temperature zero with extended reasoning disabled.

most informative, and disabled for the rest. On the 50 projects, the classifier reaches 90% category accuracy and 92% accuracy on the enable-or-disable decision.

A.5.5 Model Hyperparameters

Table A.17 lists the hyperparameters for all six configurations. All coding models use temperature zero. The GLM-5 diagnostic model is accessed with reasoning disabled; the GPT-5.4 diagnostic model uses a low reasoning-effort setting.

A.5.6 Diagnostic Overhead

Table A.18 reports how often the diagnostic scripts ran successfully. Terminal scripts succeed more often than browser scripts (89% against 55%) because they do not depend on a running dev server. Across the 66 retried tasks, the diagnostic agents made 95 successful calls, against 2,841 coding-model calls for the same tasks, which

Category	Retried	Browser	Terminal	Either
Standards	44	33 (75%)	37 (84%)	37 (84%)
UI	15	1 (7%)	15 (100%)	15 (100%)
State	7	2 (29%)	7 (100%)	7 (100%)
Total	66	36 (55%)	59 (89%)	59 (89%)

Table A.18: Diagnostic-script execution on the enabled-category retried tasks (C2). Each cell is the fraction of retried tasks for which that agent produced at least one valid observation.

is 3.3% of all language-model calls and under 2% of the total cost; the two scripts run in parallel in about 15 seconds.

A.5.7 Single-Shot Design: Why Not Interpret the Observations

An earlier two-phase version of WatchPoint ran a fixed script to collect observations and then sent them to a language model to interpret before passing the result to the coding agent. The interpretation step hallucinated. On one calculator task, the coding model's first attempt was nearly correct, but the raw script output (taken from a stale port) read:

```
Page loaded: Blog
Found 0 buttons
Console errors: ["504 Outdated Optimize Dep"]
```

and the interpreting model turned this into "the wrong page loaded, the page displays Blog content, the M+ button is missing, a 504 error indicates critical server failure". Every claim was false: the calculator is a single file with no blog page, and the 504 was a transient dev-server warning. Acting on it, the coding model abandoned its nearly correct code and spent more than 40 turns chasing phantom problems. The single-shot design avoids this by passing raw facts straight through:

```
CHECK button_count: 22
CHECK clear_button_exists: true
```

Project	C1 P@1	C2 P@1	Tasks flipped
table	30%	0%	7
angular	25%	0%	6
react	0%	25%	6
fastify	25%	0%	5
sequelize	50%	25%	5

Table A.19: Largest first-attempt pass-rate differences between C1 and C2, which share the same coding model at temperature zero. Because the simulated user cannot affect the first attempt, every difference here is run-to-run variance.

```
3 CHECK clear_button_style: "grid-column: 1 / span 3"
```

On the calculator project, the interpreting version cut the first-attempt pass count from 11 of 20 to 4 of 20; removing the interpretation step restored a net-positive effect. This is the basis for the design principle that the model decides what to measure but never interprets what was measured.

A.5.8 Run-to-Run Variance

All experiments use temperature zero, yet there is real run-to-run variance. A clean way to see it: because the simulated user only fires after the first attempt fails, the first-attempt pass rate of a baseline (C1) and its simulated-user counterpart (C2) should match if there were no noise, so any difference is pure variance. Across the 50 projects, 27 (54%) show such a difference, and 113 of 980 tasks (11.5%) flip between pass and fail across the two runs. Table A.19 lists the largest cases. The variance comes from the non-determinism of agentic tool use, not from model sampling: dev servers start at variable speed, tool calls occasionally time out under network latency, and directory listings return in different orders. The sequential stopping rule then amplifies a single early difference into a large per-project swing. This is why the chapter reports per-category results (with the disabled categories as a variance control),

Category	Project	WatchPoint	C1 P@2	C2 P@2	Δ
Standards	esmodule	ON	0	9	+9
	dom	ON	3	8	+5
	typescript	ON	5	9	+4
	form	ON	1	5	+4
	flex	ON	8	10	+2
	selector	ON	6	8	+2
	svg-chart	ON	0	2	+2
	bom	ON	4	5	+1
	float	ON	1	2	+1
	svg-solar	ON	1	2	+1
	canvas	ON	1	1	0
	chart	ON	6	6	0
	dom1	ON	3	3	0
	survey	ON	5	5	0
	sass	ON	7	5	−2
	grid	ON	10	6	−4
	table	ON	8	0	−8
	draw	ON	15	0	−15
UI	react	ON	0	7	+7
	threejs	ON	1	6	+5
	react-no-ts	ON	5	8	+3
	vue	ON	9	8	−1
	angular	ON	7	0	−7
State	zustand	ON	1	7	+6
	mobx	ON	2	3	+1
	jotai	ON	0	0	0
	redux	ON	6	5	−1
Other	expression-editor	OFF	7	14	+7
	calculator	OFF	10	12	+2
	calculator-files	OFF	10	10	0

Table A.20: Per-project Pass@2 (tasks out of 20) for the MiniMax M2.7 baseline (C1) and MiniMax M2.7 with a GLM-5 simulated user (C2), sorted by the change within each category.

the per-task recovery rate (which the cascade does not distort), and the per-project breakdown below.

A.5.9 Per-Project Results

Table A.20 gives the per-project Pass@2 for the baseline and the simulated-user configuration. The largest gains come from early-task recoveries that unblock a chain of

later tasks (for example, esmodule goes from 0 to 9). The largest drops (draw, table, angular) trace to first-attempt variance rather than to the feedback, as the variance control above shows.

A.5.10 User Study Protocol

Two graduate students with web-development experience served as human testers, each evaluating five projects under the same MiniMax M2.7 coding model and OpenHands scaffold as configuration C1. The only change from C1 is the source of the retry feedback: instead of the automated observations, the participant interacted with the live application through a browser and typed free-form feedback, which was appended to the retry prompt. To keep the comparison fair, the coding model, temperature, turn budget, warm-start retry, and stopping rule were all held constant. Each session lasted about two hours, and participants were compensated with a meal or an equivalent gift card and gave informed consent. Table A.21 lists every feedback message and whether the task was recovered. The pattern is that specific feedback recovers tasks (the precise tester reached 62%) while vague feedback does not (the other reached 44%), which is the complementary-strengths point made in the chapter.

A.5.11 Diagnostic Model Comparison

The two diagnostic models, GLM-5 (C2) and GPT-5.4 (C3) on the MiniMax M2.7 coding model, reach the same aggregate Pass@2 but help on different projects: GLM-5 does better on Standards projects with styling and structure issues, while GPT-5.4 does better on state and vector-graphics projects. Many of the per-project differences between them are first-attempt variance rather than genuine differences in diagnostic quality, and the cascade metric cancels much of the rest. The lesson is that diagnostic quality depends on the type of failure, not on the general capability of the diagnostic

Who	Task	Feedback	Rec.?
A	calc-7	“I think it’s fine”	Yes
A	calc-11	“I think it’s fine.”	No
A	chart-1	“the UI fails to render the chart entirely, which violates the requirement.”	Yes
A	chart-5	“the UI fails to display the line chart and its data points.”	No
A	expr-1	“the site cannot be reached”	No
A	flex-8	“Item 1, Item 2, and Item 3 are still in a horizontal row on the right, not stacking vertically”	No
A	form-2	“the Task 1 empty question (title: ‘sample empty question’) is completely missing from the UI.”	Yes
A	form-4	“this site can’t be reached”	Yes
A	form-5	“only a submit button showed in the UI.”	No
B	calc-5	“it looks like correct”	No
B	chart-1	“The controls render, but toggling Axes does not correctly recreate and redraw the chart in #chart.”	Yes
B	chart-6	“Changing pointStyle does not correctly update the chart points; shapes are not rendered or updated.”	Yes
B	chart-8	“Selecting SmoothLine Chart does not correctly render smooth curved dataset lines.”	No
B	expr-1	“retry”	No
B	flex-7	“On small screens ($\leq$399px), the rightbar is not stacked at the bottom of main as required.”	Yes
B	flex-9	“The leftbar items do not fully fill the 20$\times$2 layout; there is still remaining vertical space.”	Yes
B	flex-11	“On screens under 400px, the rightbar does not correctly limit its display to the first 3 rows.”	Yes
B	flex-12	“The content area does not correctly show 12 cards in a 3-per-row layout with vertical scrolling.”	No
B	form-2	“The radio options render, but the single-selection question is not fully implemented as required.”	Yes
B	form-4	“The open questions are missing; neither the single-line nor multiline input is rendered.”	Yes
B	form-5	“Stars render, but clicking them does not fully update the rating state as required.”	Yes
B	form-9	“The contents panel is not fixed on the right side.”	No

Table A.21: All human feedback texts from the user study. “Rec.?” marks whether the task was recovered (the first attempt failed and the second passed after the feedback). Participants are anonymized as A and B.

model.

Plain name	Chapter	Canonical identifier	Serving provider	Accessed
FewVuln (Chapter 3): named entity recognition backbones.				
BERT	3	`bert-large-cased`	HuggingFace, local GPU	2022
RoBERTa	3	`roberta-large`	HuggingFace, local GPU	2022
ELECTRA	3	`electra-base-discriminator`	HuggingFace, local GPU	2022
TestAug (Chapter 4): test-case generator.				
GPT-3	4	`text-davinci-003`	OpenAI API	2022
PatchHolmes (Chapter 5): agent backbones and retrievers.				
Qwen3-235B	5	`Qwen/Qwen3-235B-A22B-Instruct-2507`	OpenRouter and in-house vLLM	2026-05-01
Qwen3-Coder-30B	5	`Qwen/Qwen3-Coder-30B-A3B-Instruct`	in-house vLLM	2026-05-01
Qwen3-Embedding-8B	5	`Qwen/Qwen3-Embedding-8B`	HuggingFace, local GPU	2026-05-01
e5	5	`intfloat/e5-base-v2`	HuggingFace, local GPU	2026-05-01
WatchPoint (Chapter 6): coding and diagnostic models.				
MiniMax M2.7	6	coding model	MiniMax API	2026
GLM-5	6	coding and diagnostic model	OpenRouter (coding), Z.ai (diagnostic)	2026
GPT-5.4	6	diagnostic model	OpenAI API	2026
Sonnet 4.6	6	coding model	Anthropic (Claude Code)	2026
Haiku 4.5	6	diagnostic model	Anthropic	2026

Table A.22: Canonical provider identifiers for the models used in this dissertation. The main text refers to each model by its plain name; this table is the single place the exact identifiers appear. For the WatchPoint models (Chapter 6), the exact API checkpoint strings were not recorded in the source, so the table lists each model's role instead.

A.6 Model Identifiers Across Chapters

Table A.22 lists, for every model used in the dissertation, the plain name used in the main text, the chapter that uses it, the exact provider identifier, the serving provider, and the access date.

Bibliography

Antypas, D. and J. Camacho-Collados (2023). Robust Hate Speech Detection in Social Media: A Cross-Dataset Empirical Evaluation. In *The 7th Workshop on Online Abuse and Harms (WOAH)*, Toronto, Canada, pp. 231–242. Association for Computational Linguistics. (Pages 63, 65, 16, 17, and 18)

Asai, A., Z. Wu, Y. Wang, A. Sil, and H. Hajishirzi (2023, October). Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection. (Pages 72 and 84)

Bai, B., J. Liang, G. Zhang, H. Li, K. Bai, and F. Wang (2021). Why Attentions May Not Be Interpretable? *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. (Page 52)

Balog, K. and C. Zhai (2025, January). User Simulation in the Era of Generative AI: User Modeling, Synthetic Data Generation, and System Evaluation. (Pages 15 and 26)

Basile, V., C. Bosco, E. Fersini, D. Nozza, V. Patti, F. M. Rangel Pardo, P. Rosso, and M. Sanguinetti (2019). SemEval-2019 Task 5: Multilingual Detection of Hate Speech Against Immigrants and Women in Twitter. In *Proceedings of the International Workshop on Semantic Evaluation*. (Pages 17 and 23)

Bhandari, G. P., A. Naseer, and L. Moonen (2021). CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. *arXiv preprint arXiv:2107.08760*. (Page 21)

Bowman, S. R., G. Angeli, C. Potts, and C. D. Manning (2015). A Large Annotated Corpus for Learning Natural Language Inference. In *EMNLP*. (Page 57)

Bridges, R. A., C. L. Jones, M. D. Iannacone, K. M. Testa, and J. R. Goodall (2013). Automatic Labeling for Entity Extraction in Cyber Security. *arXiv preprint arXiv:1308.4941*. (Pages 19, 20, and 32)

Brown, T. B., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei (2020, July). Language Models are Few-Shot Learners. (Pages 24, 33, 53, and 68)

Cao, R., M. Chen, J. Chen, Z. Cui, Y. Feng, B. Hui, Y. Jing, K. Li, M. Li, J. Lin, Z. Ma, K. Shum, X. Wang, J. Wei, J. Yang, J. Zhang, L. Zhang, Z. Zhang, W. Zhao, and F. Zhou (2026, February). Qwen3-Coder-Next Technical Report. (Pages 82 and 33)

Cardiff NLP (2023). CardiffNLP twitter-RoBERTa-base-hate model. (Page 16)

Cemri, M., M. Z. Pan, S. Yang, L. A. Agrawal, B. Chopra, R. Tiwari, K. Keutzer, A. Parameswaran, D. Klein, K. Ramchandran, M. Zaharia, J. E. Gonzalez, and I. Stoica (2025, October). Why Do Multi-Agent LLM Systems Fail? (Page 86)

Chang, J.-R. and Y.-S. Chen (2018). Pyramid Stereo Matching Network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5410–5418. (Page 33)

Chang, S., A. Anderson, and J. M. Hofman (2025). ChatBench: From Static Benchmarks to Human-AI Evaluation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 26009–26038. (Pages 15 and 106)

Chen, H., M. Xia, and D. Chen (2021). Non-Parametric Few-Shot Learning for Word Sense Disambiguation. *arXiv preprint arXiv:2104.12677*. (Page 33)

Chen, M., L. Sun, T. Li, H. Sun, Y. Zhou, C. Zhu, H. Wang, J. Z. Pan, W. Zhang, H. Chen, F. Yang, Z. Zhou, and W. Chen (2025). ReSearch: Learning to Reason with Search for Llms Via Reinforcement Learning. (Page 84)

Chen, M., J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba (2021, July). Evaluating Large Language Models Trained on Code. (Page 89)

Chen, X., M. Lin, N. Schärli, and D. Zhou (2023, October). Teaching Large Language Models to Self-Debug. (Page 87)

Chen, Z., X. Ma, S. Zhuang, P. Nie, K. Zou, A. Liu, J. Green, K. Patel, R. Meng, M. Su, S. Sharifymoghaddam, Y. Li, H. Hong, X. Shi, X. Liu, N. Thakur, C. Zhang, L. Gao, W. Chen, and J. Lin (2025, August). BrowseComp-Plus: A More Fair and Transparent Evaluation Benchmark of Deep-Research Agent. (Page 83)

Clark, K., M.-T. Luong, Q. V. Le, and C. D. Manning (2020, March). ELECTRA: Pre-Training Text Encoders as Discriminators Rather Than Generators. (Page 37)

Crispin, L. and J. Gregory (2014). *Agile Testing: A Practical Guide for Testers and Agile Teams* (1. ed., 10. print ed.). The Addison-Wesley Signature Series. Upper Saddle River: Addison-Wesley. (Page 11)

Croft, R., M. A. Babar, and M. Kholoosi (2023, January). Data Quality for Software Vulnerability Datasets. (Pages 18, 19, and 20)

Davidson, T., D. Warmsley, M. Macy, and I. Weber (2017, March). Automated Hate Speech Detection and the Problem of Offensive Language. (Pages 68 and 17)

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova (2019, May). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. (Pages 22, 37, 49, and 71)

Dijkstra, E. W. (1972). Notes on Structured Programming. In *Structured Programming*, pp. 1–82. Academic Press. (Page 10)

Dissanayake, N., A. Jayatilaka, M. Zahedi, and M. A. Babar (2021, August). Software Security Patch Management – A Systematic Literature Review of Challenges, Approaches, Tools and Practices. (Pages 1 and 4)

Dodge, J., G. Ilharco, R. Schwartz, A. Farhadi, H. Hajishirzi, and N. A. Smith (2020). Fine-Tuning Pretrained Language Models: Weight Initializations, Data Orders, and Early Stopping. *ArXiv abs/2002.06305*. (Page 20)

Dong, Y., W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang (2019). Towards the Detection of Inconsistencies in Public Security Vulnerability Reports. In *28th USENIX Security Symposium (USENIX Security 19)*, pp. 869–885. (Pages 19, 20, 32, 33, 34, 36, 40, and 48)

Dunlap, T., E. Lin, W. Enck, and B. Reaves (2024, July). VFCFinder: Pairing Security Advisories and Patches. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, New York, NY, USA, pp. 1128–1142. Association for Computing Machinery. (Pages 70 and 72)

ElSherief, M., C. Ziems, D. Muchlinski, V. Anupindi, J. Seybolt, M. De Choudhury, and D. Yang (2021). Latent Hatred: A Benchmark for Understanding Implicit Hate Speech. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Online and Punta Cana, Dominican Republic, pp. 345–363. Association for Computational Linguistics. (Page 68)

Facebook (2021). Facebook RoBERTa Model R1 for Hate Speech. (Pages 63 and 16)

Facebook (2022). Facebook Community Standards on Hate Speech. (Pages xiv, 54, 62, 68, and 23)

Fan, A., B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang (2023, November). Large Language Models for Software Engineering: Survey and Open Problems. (Pages 17 and 18)

Farhang, S., M. B. Kirdan, A. Laszka, and J. Grossklags (2020). An Empirical Study of Android Security Bulletins in Different Vendors. In *Proceedings of the Web Conference 2020*, pp. 3063–3069. (Pages 18 and 32)

Feng, S., C. Du, H. Liu, Q. Wang, Z. Lv, G. Huo, X. Yang, and C. Chen (2025, April). Agent for User: Testing Multi-User Interactive Features in TikTok. (Page 15)

Finn, C., P. Abbeel, and S. Levine (2017). Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *International Conference on Machine Learning*, pp. 1126–1135. PMLR. (Pages 33 and 49)

Founta, A., C. Djouvas, D. Chatzakou, I. Leontiadis, J. Blackburn, G. Stringhini, A. Vakali, M. Sirivianos, and N. Kourtellis (2018). Large Scale Crowdsourcing and Characterization of Twitter Abusive Behavior. In *Proceedings of the International Conference on Web and Social Media*. (Page 18)

Friedl, P. (2023). Dis/Similarities in the Design and Development of Legal and Algorithmic Normative Systems: The Case of Perspective API. *Law, Innovation and Technology*. (Page 15)

Fritzler, A., V. Logacheva, and M. Kretov (2019). Few-Shot Classification in Named Entity Recognition Task. *Proceedings of the 34th ACM/SIGAPP Symposium on*

Applied Computing. (Page 20)

Gaglione Jr, G. S. (2019). The Equifax Data Breach: An Opportunity to Improve Consumer Protection and Cybersecurity Efforts in America. *67*, 1133. (Pages 18 and 31)

Gandhi, S., J. Tsay, J. Ganhotra, K. Kate, and Y. Rizk (2025, September). When Agents go Astray: Course-Correcting SWE Agents with PRMs. (Pages 87, 92, 96, and 100)

Gao, P., X. Liu, E. Choi, B. Soman, C. Mishra, K. Farris, and D. Song (2021). A System for Automated Open-Source Threat Intelligence Gathering and Management. *arXiv preprint arXiv:2101.07769*. (Pages 20 and 32)

Gao, T., A. Fisch, and D. Chen (2021, June). Making Pre-trained Language Models Better Few-shot Learners. (Pages 24, 33, and 56)

Gasmi, H., J. Laval, and A. Bouras (2019). Information Extraction of Cybersecurity Concepts: An LSTM Approach. *Applied Sciences 9*(19), 3945. (Pages 20, 32, and 33)

Geva, M., Y. Goldberg, and J. Berant (2019). Are We Modeling the Task or the Annotator? An Investigation of Annotator Bias in Natural Language Understanding Datasets. *ArXiv abs/1908.07898*. (Pages 13 and 52)

Glockner, M., V. Shwartz, and Y. Goldberg (2018). Breaking NLI Systems with Sentences That Require Simple Lexical Inferences. In *ACL*. (Page 52)

Google (2023a). Google Perspective API. (Pages 63 and 15)

Google (2023b). A List of Publishers Using the Perspective API. (Page 15)

Grimminger, L. and R. Klinger (2021, March). Hate Towards the Political Opponent: A Twitter Corpus Study of the 2020 US Elections on the Basis of Offensive Speech and Stance Detection. (Pages 17 and 23)

Grishman, R. and B. Sundheim (1996). Message Understanding Conference- 6: A Brief History. In *COLING*. (Page 49)

Gröndahl, T., L. Pajola, M. Juuti, M. Conti, and N. Asokan (2018). All You Need Is "Love": Evading Hate Speech Detection. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security*. (Page 68)

Guo, J., J. Cheng, and J. Cleland-Huang (2017, May). Semantically Enhanced Software Traceability Using Deep Learning Techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 3–14. (Page 17)

Gururangan, S., S. Swayamdipta, O. Levy, R. Schwartz, S. R. Bowman, and N. A. Smith (2018). Annotation Artifacts in Natural Language Inference Data. In *NAACL*. (Pages 13 and 52)

Hartvigsen, T., S. Gabriel, H. Palangi, M. Sap, D. Ray, and E. Kamar (2022, July). ToxiGen: A Large-Scale Machine-Generated Dataset for Adversarial and Implicit Hate Speech Detection. (Page 68)

He, B., C. Ziems, S. Soni, N. Ramakrishnan, D. Yang, and S. Kumar (2021, November). Racism is a Virus: Anti-Asian Hate and Counterspeech in Social Media during the COVID-19 Crisis. (Page 68)

He, J., C. Treude, and D. Lo (2025, May). LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. *ACM Trans. Softw. Eng. Methodol. 34*(5), 124:1–124:30. (Pages 25 and 86)

Hoffmann, J., S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre (2022, March). Training Compute-Optimal Large Language Models. (Page 24)

Hou, Y., W. Che, Y. Lai, Z. Zhou, Y. Liu, H. Liu, and T. Liu (2020). Few-Shot Slot Tagging with Collapsed Dependency Transfer and Label-Enhanced Task-Adaptive Projection Network. *arXiv preprint arXiv:2006.05702*. (Page 33)

Huang, J., C. Li, K. Subudhi, D. Jose, S. Balakrishnan, W. Chen, B. Peng, J. Gao, and J. Han (2020, December). Few-Shot Named Entity Recognition: A Comprehensive Study. (Page 20)

Huyen, C. (2022). *Designing Machine Learning Systems: An Iterative Process for Production-Ready Applications* (First edition ed.). Beijing Boston Farnham Sebastopol Tokyo: O'Reilly. (Page 12)

Jiang, J. A., S. Middler, J. R. Brubaker, and C. Fiesler (2020). Characterizing Community Guidelines on Social Media Platforms. In *Conference Companion Publication of the Computer Supported Cooperative Work and Social Computing*. (Page 62)

Jiang, Z., F. F. Xu, L. Gao, Z. Sun, Q. Liu, J. Dwivedi-Yu, Y. Yang, J. Callan, and G. Neubig (2023, October). Active Retrieval Augmented Generation. (Pages 72 and 84)

Jimenez, C. E., J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan (2023, October). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? (Pages 15, 26, 89, and 99)

Jin, B., H. Zeng, Z. Yue, J. Yoon, S. Arik, D. Wang, H. Zamani, and J. Han (2025, August). Search-R1: Training LLMs to Reason and Leverage Search Engines with Reinforcement Learning. (Pages 72 and 84)

Jin, J., Y. Zhu, Z. Dou, G. Dong, X. Yang, C. Zhang, T. Zhao, Z. Yang, and J.-R. Wen (2025, May). FlashRAG: A Modular Toolkit for Efficient Retrieval-Augmented Generation Research. In *Companion Proceedings of the ACM on Web Conference 2025*, WWW '25, New York, NY, USA, pp. 737–740. Association for Computing Machinery. (Pages 79 and 34)

Joshi, P., S. Aditya, A. Sathe, and M. Choudhury (2020, October). TaxiNLI: Taking a Ride up the NLU Hill. (Pages xiii and 6)

Jurafsky, D. and J. H. Martin (2000). Speech and Language Processing. (Page 58)

Kaplan, J., S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei (2020, January). Scaling Laws for Neural Language Models. (Page 24)

Kästner, C. (2025). *Machine Learning in Production: From Models to Products* (First edition ed.). Cambridge, Massachusetts: The MIT Press. (Page 12)

Koh, J. Y., R. Lo, L. Jang, V. Duvvur, M. C. Lim, P.-Y. Huang, G. Neubig, S. Zhou, R. Salakhutdinov, and D. Fried (2024, June). VisualWebArena: Evaluating Multimodal Agents on Realistic Visual Web Tasks. (Page 26)

Krasner, H. (2022, December). The Cost of Poor Software Quality in the US: A 2022 Report. Technical report, Consortium for Information & Software Quality (CISQ). (Page 1)

Le, T. H. M., H. Chen, and M. A. Babar (2023, May). A Survey on Data-driven Software Vulnerability Assessment and Prioritization. *ACM Computing Surveys 55*(5), 1–39. (Pages 4, 18, 19, 20, and 21)

Li, J., B. Chiu, S. Feng, and H. Wang (2020). Few-Shot Named Entity Recognition Via Meta-Learning. *IEEE Transactions on Knowledge and Data Engineering*, 1–1. (Page 20)

Li, J., A. Sun, J. Han, and C. Li (2022, January). A Survey on Deep Learning for Named Entity Recognition. *IEEE Transactions on Knowledge and Data Engineering 34*(1), 50–70. (Page 23)

Li, K., J. Zhang, S. Chen, H. Liu, Y. Liu, and Y. Chen (2024, September). PatchFinder: A Two-Phase Approach to Security Patch Tracing for Disclosed Vulnerabilities in Open-Source Software. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 590–602. (Pages 70, 72, 80, and 83)

Li, Y., C. Zhang, R. Lv, A. Liu, K. Deng, Y. Zhang, J. Liu, W. Zhou, and B. Zhou (2025, October). ReLook: Vision-Grounded RL with a Multimodal LLM Critic for Agentic Web Coding. (Pages 87 and 92)

Li, Z., D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen (2022, July). SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing 19*(4), 2244–2258. (Page 18)

Liu, A., S. Swayamdipta, N. A. Smith, and Y. Choi (2022, November). WANLI: Worker and AI Collaboration for Natural Language Inference Dataset Creation. (Page 55)

Liu, C., Z. Gu, G. Wu, Y. Zhang, J. Wei, and T. Xie (2025, May). Temac: Multi-Agent Collaboration for Automated Web GUI Testing. (Page 15)

Liu, J., K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou (2025, December). Large Language Model-Based Agents for Software Engineering: A Survey. (Pages 3, 4, 5, and 105)

Liu, P., W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig (2021). Pre-Train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ArXiv abs/2107.13586*. (Page 24)

Liu, X., J. Zheng, G. Yang, S. Wen, Q. Liu, and X. Wang (2025, August). Fast and Accurate Silent Vulnerability Fix Retrieval. (Pages xx, 70, 71, 72, 73, 74, 75, 78, and 83)

Liu, Y., M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov (2019, July). RoBERTa: A Robustly Optimized BERT

Pretraining Approach. (Pages 22, 37, and 49)

Lu, P., S. Zhang, Y. Hou, L. Ye, C. Huang, Z. Chen, J. Zeng, H. Jiang, P. Liu, Y. Wang, and M.-H. Yang (2026, February). ProjDevBench: Benchmarking AI Coding Agents on End-to-End Project Development. (Page 14)

Lu, Y., B. Yao, H. Gu, J. Huang, Z. J. Wang, Y. Li, J. Gesi, Q. He, T. J.-J. Li, and D. Wang (2025, April). UXAgent: An LLM Agent-Based Usability Testing Framework for Web Design. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA '25, New York, NY, USA, pp. 1–12. Association for Computing Machinery. (Page 15)

Lu, Z., H. Ren, Y. Yang, K. Wang, Z. Zong, J. Pan, M. Zhan, and H. Li (2025, September). WebGen-Agent: Enhancing Interactive Website Generation with Multi-Level Feedback and Step-Level Reinforcement Learning. (Pages 87, 92, and 100)

Ma, Z., C. Peng, Q. Zeng, P. Gao, Y. Zou, and B. Xie (2025, August). Tool-integrated Reinforcement Learning for Repo Deep Search. (Page 84)

Maddila, C., A. Tait, C. Chang, D. Cheng, N. Ahmad, V. Murali, M. Roch, A. Avondet, A. Meltzer, V. Montalvao, M. Hopko, C. Waterson, P. Thakkar, R. Fernandez, K. Kristensen, S. Barzily, S. Chen, R. Abreu, N. Nagappan, P. Shodjai, K. Murphy, J. Everingham, A. Ramani, and P. C. Rigby (2025). Agentic Program Repair from Test Failures at Scale: A Neuro-symbolic approach with static analysis and test execution feedback. (Page 87)

Magar, I. and R. Schwartz (2022). Data Contamination: From Memorization to Exploitation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. (Page 16)

Markov, T., C. Zhang, S. Agarwal, T. Eloundou, T. Lee, S. Adler, A. Jiang, and L. Weng (2023, February). A Holistic Approach to Undesired Content Detection in the Real World. (Page 16)

Mathew, B., P. Saha, S. M. Yimam, C. Biemann, P. Goyal, and A. Mukherjee (2022, April). HateXplain: A Benchmark Dataset for Explainable Hate Speech Detection. (Pages 68, 17, and 23)

McHugh, M. L. (2012). Interrater Reliability: The Kappa Statistic. *Biochemia Medica 22*, 276–282. (Page 61)

MITRE Corporation (2024). Common Vulnerabilities and Exposures. (Page 1)

Mulwad, V., W. Li, A. Joshi, T. Finin, and K. Viswanathan (2011). Extracting Information About Security Vulnerabilities from Web Text. In *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, Volume 3, pp. 257–260. IEEE. (Page 32)

Myers, G. J. (2012). *The Art of Software Testing* (3rd ed ed.). Hoboken, N.J: J. Wiley & Sons. (Pages 9, 10, 11, and 12)

National Institute of Standards and Technology (2024). National Vulnerability Database. (Page 1)

O'Donoghue, E., B. Boles, C. Izurieta, and A. M. Reinhold (2024, November). Impacts of Software Bill of Materials (SBOM) Generation on Vulnerability Detection.

In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, SCORED '24, New York, NY, USA, pp. 67–76. Association for Computing Machinery. (Page 70)

Omri, S. and C. Sinz (2021, April). Machine Learning Techniques for Software Quality Assurance: A Survey. (Page 1)

OpenAI (2023). OpenAI Moderation API. (Page 63)

Ousidhoum, N., Z. Lin, H. Zhang, Y. Song, and D.-Y. Yeung (2019). Multilingual and Multi-Aspect Hate Speech Analysis. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Hong Kong, China, pp. 4674–4683. Association for Computational Linguistics. (Page 17)

Ouyang, L., J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe (2022, March). Training Language Models to Follow Instructions with Human Feedback. (Page 25)

Palmer, C. (2020, February). The Boeing 737 Max Saga: Automating Failure. *Engineering 6*(1), 2–3. (Page 31)

Park, J. S., J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein (2023, August). Generative Agents: Interactive Simulacra of Human Behavior. (Pages 15 and 26)

Phang, J., T. Févry, and S. R. Bowman (2018). Sentence Encoders on Stilts: Supplementary Training on Intermediate Labeled-Data Tasks. *arXiv preprint arXiv:1811.01088*. (Page 38)

Philipov, D., V. Dongre, G. Tur, and D. Hakkani-Tür (2024, October). Simulating User Agents for Embodied Conversational-AI. (Page 26)

Poliak, A., J. Naradowsky, A. Haldar, R. Rudinger, and B. Van Durme (2018). Hypothesis Only Baselines in Natural Language Inference. In *Proceedings of the Seventh Joint Conference on Lexical and Computational Semantics*. (Page 68)

Qian, C., W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun (2024, June). ChatDev: Communicative Agents for Software Development. (Page 86)

Qiu, X., T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang (2020, March). Pre-Trained Models for Natural Language Processing: A Survey. (Pages 22 and 23)

Radford, A. and K. Narasimhan (2018). Improving Language Understanding by Generative Pre-Training. (Page 22)

Rahman, M. M., D. Balakrishnan, D. Murthy, M. Kutlu, and M. Lease (2021, November). An Information Retrieval Approach to Building Datasets for Hate Speech Detection. (Page 68)

Rajpurkar, P., J. Zhang, K. Lopyrev, and P. Liang (2016, October). SQuAD: 100,000+ Questions for Machine Comprehension of Text. (Page 68)

Reif, E., D. Ippolito, A. Yuan, A. Coenen, C. Callison-Burch, and J. Wei (2022,

March). A Recipe For Arbitrary Text Style Transfer with Large Language Models. (Page 55)

Reynolds, L. and K. McDonell (2021). Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. (Pages 24 and 55)

Ribeiro, M. T. and S. Lundberg (2022). Adaptive Testing and Debugging of NLP Models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Dublin, Ireland, pp. 3253–3267. Association for Computational Linguistics. (Page 14)

Ribeiro, M. T., T. Wu, C. Guestrin, and S. Singh (2020, May). Beyond Accuracy: Behavioral Testing of NLP models with CheckList. (Pages xiii, 13, 14, 52, 53, 54, 55, 68, and 6)

Rogers, A., O. Kovaleva, and A. Rumshisky (2020, November). A Primer in BERTology: What We Know About How BERT Works. (Pages 22 and 23)

Röttger, P., B. Vidgen, D. Nguyen, Z. Waseem, H. Margetts, and J. B. Pierrehumbert (2021). HateCheck: Functional Tests for Hate Speech Detection Models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 41–58. (Pages 14, 66, 68, and 23)

Sachdeva, P., R. Barreto, G. Bacon, A. Sahn, C. von Vacano, and C. Kennedy (2022). The Measuring Hate Speech Corpus: Leveraging Rasch Measurement Theory for Data Perspectivism. In *Proceedings of the Workshop on Perspectivist Approaches to NLP @LREC2022*. (Pages 68 and 17)

Samory, M., I. Sen, J. Kohne, F. Floeck, and C. Wagner (2021, June). "Call me sexist, but...": Revisiting Sexism Detection Using Psychological Scales and Adversarial Samples. (Page 17)

Schick, T. and H. Schütze (2021). Generating Datasets with Pretrained Language Models. In *EMNLP*. (Page 55)

Seshadri, P., S. Cahyawijaya, A. Odumakinde, S. Singh, and S. Goldfarb-Tarrant (2026, January). Lost in Simulation: LLM-Simulated Users are Unreliable Proxies for Human Users in Agentic Evaluations. (Page 92)

Shah, P., R. Ghosh, A. Singhal, and D. Dutta (2025, September). RANGER – Repository-Level Agent for Graph-Enhanced Retrieval. (Page 84)

Singh, R., S. Joel, A. Mehrotra, N. Wadhwa, R. B. Bairi, A. Kanade, and N. Natarajan (2026, May). Code Researcher: Deep Research Agent for Large Systems Code and Commit History. (Page 84)

Snell, J., K. Swersky, and R. S. Zemel (2017). Prototypical Networks for Few-Shot Learning. *arXiv preprint arXiv:1703.05175*. (Pages 20, 33, and 49)

Socher, R., A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts (2013, October). Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, Seattle, Washington, USA, pp. 1631–1642. Association for

Computational Linguistics. (Page 57)

Storhaug, A., J. Sun, and J. Li (2026, February). Favia: Forensic Agent for Vulnerability-fix Identification and Analysis. (Pages xii, xx, 70, 72, 73, 74, 78, 79, 80, 83, 84, and 33)

Su, H., H. Yen, M. Xia, W. Shi, N. Muennighoff, H.-y. Wang, H. Liu, Q. Shi, Z. S. Siegel, M. Tang, R. Sun, J. Yoon, S. O. Arik, D. Chen, and T. Yu (2024, July). BRIGHT: A Realistic and Challenging Benchmark for Reasoning-Intensive Retrieval. (Page 83)

Sun, J., Z. Xing, H. Guo, D. Ye, X. Li, X. Xu, and L. Zhu (2021). Generating Informative CVE Description from ExploitDB Posts by Extractive Summarization. *arXiv preprint arXiv:2101.01431*. (Page 32)

Sun, L., S. Fu, B. Yao, Y. Lu, W. Li, H. Gu, J. Gesi, J. Huang, C. Luo, and D. Wang (2025, September). LLM Agent Meets Agentic AI: Can LLM Agents Simulate Customers to Evaluate Agentic-AI-based Shopping Assistants? (Page 100)

Talat, Z. and D. Hovy (2016). Hateful Symbols or Hateful People? Predictive Features for Hate Speech Detection on Twitter. In *Proceedings of the NAACL Student Research Workshop*. (Pages 68 and 18)

Tan, X., Y. Zhang, C. Mi, J. Cao, K. Sun, Y. Lin, and M. Yang (2021, November). Locating the Security Patches for Disclosed OSS Vulnerabilities with Vulnerability-Commit Correlation Ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, New York, NY, USA, pp. 3282–3299. Association for Computing Machinery. (Pages 70, 71, 72, and 83)

Taori, R., I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto (2023). Stanford Alpaca Model. (Page 16)

Tarunesh, I., S. Aditya, and M. Choudhury (2021). LoNLI: An Extensible Framework for Testing Diverse Logical Reasoning Capabilities for NLI. In *Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI-22)*, Volume abs/2112.02333. (Pages xiii, 52, 53, and 6)

Tjong Kim Sang, E. F. and F. De Meulder (2003). Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003 - Volume 4*, Conll '03, USA, pp. 142–147. Association for Computational Linguistics. (Page 33)

Trivedi, H., N. Balasubramanian, T. Khot, and A. Sabharwal (2023, June). Interleaving Retrieval with Chain-of-Thought Reasoning for Knowledge-Intensive Multi-Step Questions. (Pages 72, 73, 79, 84, and 34)

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin (2017). Attention Is All You Need. In *Neural Information Processing Systems*. (Page 23)

Vidgen, B., A. Botelho, D. Broniatowski, E. Guest, M. Hall, H. Margetts, R. Tromble, Z. Waseem, and S. Hale (2020, May). Detecting East Asian Prejudice on Social Media. (Page 18)

Vidgen, B., T. Thrush, Z. Waseem, and D. Kiela (2021, June). Learning from the Worst: Dynamically Generated Datasets to Improve Online Hate Detection. (Pages 68, 16, and 18)

Wagner, N., M. Desmond, R. Nair, Z. Ashktorab, E. M. Daly, Q. Pan, M. S. Cooper, J. M. Johnson, and W. Geyer (2024, October). Black-box Uncertainty Quantification Method for LLM-as-a-Judge. (Page 27)

Wang, A., A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman (2019, February). GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. (Page 68)

Wang, J., Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang (2024, March). Software Testing with Large Language Models: Survey, Landscape, and Vision. (Page 15)

Wang, K., T. Li, X. Zhang, C. Wang, W. Sun, Y. Liu, A. Liu, X. Liu, C. Shen, and B. Shi (2026, March). Software Development Life Cycle Perspective: A Survey of Benchmarks for Code Large Language Models and Agents. (Pages 4, 5, and 9)

Wang, X., B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig (2025, April). OpenHands: An Open Platform for AI Software Developers as Generalist Agents. (Pages 16, 26, 76, 86, 87, and 90)

Wang, X., S. Rosenberg, J. Michelini, C. Smith, H. Tran, E. Nyst, R. Malhotra, X. Zhou, V. Chen, R. Brennan, and G. Neubig (2025, November). The OpenHands Software Agent SDK: A Composable and Extensible Foundation for Production Agents. (Page 26)

Wang, X., J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou (2023, March). Self-Consistency Improves Chain of Thought Reasoning in Language Models. (Page 24)

Wei, J., X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou (2023, January). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. (Page 24)

West, P., C. Bhagavatula, J. Hessel, J. D. Hwang, L. Jiang, R. L. Bras, X. Lu, S. Welleck, and Y. Choi (2022, November). Symbolic Knowledge Distillation: From General Language Models to Commonsense Models. (Page 55)

Wolf, T., L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew (2019). HuggingFace's Transformers: State-of-the-Art Natural Language Processing. *ArXiv abs/1910.03771*. (Page 38)

Wu, X., Z. Xue, D. Yin, S. Zhou, K.-W. Chang, N. Peng, and Y. Wen (2025, December). FronTalk: Benchmarking Front-End Development as Conversational Code Generation with Multi-Modal Feedback. (Pages 87 and 92)

Xia, C. S., Y. Deng, S. Dunn, and L. Zhang (2024, October). Agentless: Demystifying LLM-based Software Engineering Agents. (Pages 26 and 100)

Xu, H., S. Wang, N. Li, K. Wang, Y. Zhao, K. Chen, T. Yu, Y. Liu, and H. Wang (2025, September). Large Language Models for Cyber Security: A Systematic

Literature Review. (Pages 3 and 105)

Xu, K., Y. Mao, X. Guan, and Z. Feng (2025, May). Web-Bench: A LLM Code Benchmark Based on Web Standards and Frameworks. (Pages 15, 16, 86, 87, 89, 90, 93, 94, and 99)

Yang, G., M. Haque, Q. Song, W. Yang, and X. Liu (2022, October). TestAug: A Framework for Augmenting Capability-based NLP Tests. In N. Calzolari, C.-R. Huang, H. Kim, J. Pustejovsky, L. Wanner, K.-S. Choi, P.-M. Ryu, H.-H. Chen, L. Donatelli, H. Ji, S. Kurohashi, P. Paggio, N. Xue, S. Kim, Y. Hahm, Z. He, T. K. Lee, E. Santus, F. Bond, and S.-H. Na (Eds.), *Proceedings of the 29th International Conference on Computational Linguistics*, Gyeongju, Republic of Korea, pp. 3480–3495. International Committee on Computational Linguistics. (Page 13)

Yang, J., C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press (2024, November). SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. (Pages 78 and 100)

Yang, Y. and A. Katiyar (2020, October). Simple and Effective Few-Shot Named Entity Recognition with Structured Nearest Neighbor Learning. (Pages 33, 34, 39, and 49)

Yang, Y., X. Xia, D. Lo, and J. Grundy (2022, September). A Survey on Deep Learning for Software Engineering. *ACM Comput. Surv. 54*(10s), 206:1–206:73. (Pages 17, 18, and 20)

Yao, S., J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao (2023, March). ReAct: Synergizing Reasoning and Acting in Language Models. (Page 83)

Yitagesu, S., Z. Xing, X. Zhang, Z. Feng, X. Li, and L. Han (2023, September). Extraction of Phrase-based Concepts in Vulnerability Descriptions through Unsupervised Labeling. *ACM Transactions on Software Engineering and Methodology 32*(5), 1–45. (Pages 19 and 20)

Zampieri, M., S. Malmasi, P. Nakov, S. Rosenthal, N. Farra, and R. Kumar (2019). Predicting the Type and Target of Offensive Posts in Social Media. In *Proceedings of the 2019 Conference of the North*, Minneapolis, Minnesota, pp. 1415–1420. Association for Computational Linguistics. (Page 17)

Zhang, E., X. Wang, P. Gong, Y. Lin, and J. Mao (2024, October). USimAgent: Large Language Models for Simulating Search Users. (Page 26)

Zhang, J. M., M. Harman, L. Ma, and Y. Liu (2019). Machine Learning Testing: Survey, Landscapes and Horizons. *ArXiv abs/1906.10742*. (Page 54)

Zhang, J. M., M. Harman, L. Ma, and Y. Liu (2022). Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Transactions on Software Engineering 48*(2), 1–36. (Pages 1 and 12)

Zhang, T., F. Wu, A. Katiyar, K. Q. Weinberger, and Y. Artzi (2020, June). Revisiting Few-Sample BERT Fine-Tuning. (Page 32)

Zhang, Y., M. Li, D. Long, X. Zhang, H. Lin, B. Yang, P. Xie, A. Yang, D. Liu, J. Lin, F. Huang, and J. Zhou (2025, June). Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. (Page 75)

Zhang, Y., X. Liu, and C. Zhai (2017, October). Information Retrieval Evaluation as Search Simulation: A General Formal Framework for IR Evaluation. In *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval*, Amsterdam The Netherlands, pp. 193–200. ACM. (Page 26)

Zheng, L., W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica (2023, December). Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. (Pages 26 and 27)

Zhou, C., B. Li, and X. Sun (2020). Improving Software Bug-Specific Named Entity Recognition with Deep Neural Network. *165*, 110572. (Page 32)

Zhou, S., F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, T. Ou, Y. Bisk, D. Fried, U. Alon, and G. Neubig (2024, April). WebArena: A Realistic Web Environment for Building Autonomous Agents. (Pages 16 and 26)

Zhu, Y., S. Lu, L. Zheng, J. Guo, W. Zhang, J. Wang, and Y. Yu (2018). Texygen: A Benchmarking Platform for Text Generation Models. *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. (Page 58)

Vita

Guanqun Yang

Education

- *Stevens Institute of Technology*, Hoboken, NJ — August 2020 – July 2026
  Ph.D. in Computer Science
- *University of California*, Los Angeles, CA — September 2017 – December 2019
  M.S. in Electrical and Computer Engineering
- *Northeastern University*, Shenyang, China — September 2013 – June 2017
  B.Eng. in Electrical Engineering (Automation)

Industrial Experience

- *Capital One Financial Corporation*, McLean, VA — July 2026 – present
  Principal Data Scientist
- *Capital One Financial Corporation*, McLean, VA — June 2025 – August 2025
  Data Scientist Intern
- *Capital One Financial Corporation*, McLean, VA — June 2024 – August 2024
  Data Scientist Intern

Teaching Experience

- *Stevens Institute of Technology*, Hoboken, NJ — Fall 2020 – Spring 2024
  Teaching Assistant of Text Mining and Information Retrieval (CS-589), five semesters
- *Stevens Institute of Technology*, Hoboken, NJ
  Teaching Assistant of Algorithms (CS-590)