Evaluating and Improving the Quality of LLM-Generated Code

Tutorial @ FSE 2026 · Montreal, Canada

Glaucia Melo · Jessica Pourleyli · Genevieve Caumartin · Corey Yang-Smith · Diego Costa · Ahmad Abdellatif

Toronto Metropolitan University · Concordia University · University of Calgary

Tutorial Agenda

PART 1 · 90 min (9am to 10:30am, MB.2.435)

Foundations of LLM Code Quality

10 min	Context & Setup: LLMs in Modern SE
25 min	Maintainability & Reliability
25 min	Security Vulnerabilities
15 min	Integrated Analysis & Discussion
15 min	Bias-Aware Evaluation

☕ **BREAK**

PART 2 · 90 min (11am to 12:30pm, MB.2.435)

Remediation, Explainability & Fairness

40 min	Agentic Bug Localization
40 min	Explainability for Developer Trust
10 min	Synthesis & Human-in-the-Loop

Meet the Presenters

Jessica Pourleyli

Toronto Metropolitan University

Incoming Master's Student @ McMaster University· Security in LLM code, Vulnerability detection

Genevieve Caumartin

Concordia University

PhD Student · Context engineering, Agentic bug localization & repair

Corey Yang-Smith

University of Calgary

PhD Student · LLMs for automated SE, AI/ML team lead

Toronto Metropolitan University

Université Concordia University Montréal

University of Calgary

Why This Tutorial, Why Now?

THE QUESTION WE USED TO ASK

Can LLMs write code?

Largely answered. ~30% of new code is AI-assisted, velocity gains are real, and generation benchmarks are saturating.

THE QUESTION THAT MATTERS NOW

Can we trust what they write?

Largely open. Code that looks right can hide smells, defects, vulnerabilities, and biased evaluations of all three.

Our thesis: post-generation evaluation is a first-class software engineering problem, and today you'll leave with a working, reusable pipeline for it.

One Pipeline, Four Questions

***"Evaluating and Improving the Quality of LLM-Generated Code"**: every session today answers one piece of it.*

1
EVALUATE

Is the code maintainable, reliable, and secure?

Part 1 · Jessica
SonarQube + Bandit vs. human baselines

2
JUDGE FAIRLY

Is our evaluation itself unbiased?

Part 1 · Jessica
Bias-aware, intersectional evaluation

3
IMPROVE

Can agents localize and fix what we find?

Part 2 · Genevieve
Agentic bug localization & remediation

4
TRUST

Will a developer actually accept the fix?

Part 2 · Corey
Explainability for developer trust

Colab notebook or Menti carry you through all four: built on our peer-reviewed and under submission work across three labs, and fully reusable on your own code.

Key Learning Outcomes

By the end of this tutorial, you will be able to:

01 

Apply a reusable, end-to-end evaluation pipeline for assessing quality and security of LLM-generated code, and adapt it to new datasets, models, and tasks.

02 

Understand how explainability can create efficient software patches

03 

Apply a human-in-the-loop, agentic remediation workflow in which LLM-generated fixes are iteratively verified through automated re-analysis.

04 

Verify bias-aware analysis of contextual AI outputs in software engineering

PART 1

Foundations of LLM Code Quality + Bias Eval

90 minutes · Hands-on with Google Colab

- LLMs in Modern SE
- Maintainability & Reliability
- Security Vulnerabilities
- Integrated Analysis
- Bias-Aware Evaluation

Before We Start: Prerequisites + Task Loading

1) Scan the QR Code or visit the link below ↓

bit.ly/FSE-Maintain-Security

(link is case sensitive and requires Google Colab extension installed on Google account. If the link does not work, please access them via the google drive located at https://shorturl.at/0FvXX)

2) Make a copy of the notebook

3) Standby for further instructions! We are going to walk through the prerequisite tasks shortly!

LLMs in Modern Software Engineering

Where LLMs Are Used in Development

Code Generation

Refactoring

Test Generation

Documentation

Bug Fixing

⚠️ *The challenge: LLM-generated code can be secure-looking but semantically incorrect — requiring systematic post-generation evaluation.*

Quality & Security Risks in LLM-Generated Code

Maintainability Issues

Higher code smell density, lower cohesion, inconsistent naming conventions compared to human-written code

Functional Defects

Subtle logical errors, edge-case failures, and incorrect assumptions about library APIs or language semantics

Security Vulnerabilities

Injection flaws, insecure deserialization, hardcoded secrets — often passing superficial review

Technical Debt

Prompt-dependent quality variance makes LLM outputs unpredictable, compounding debt over time

Source: Molison et al. (ESEM 2025) · Siddiq et al. (MSR 2024) · Liu et al. (NeurIPS 2023)

The End-to-End Evaluation Pipeline

1

Code Artifacts

LLM & human-written snippets

2

Static Analysis

SonarQube + Bandit at scale

3

Metric Aggregation

Structured output, visualization, comparison

4

Fairness Evaluation

Bias detection across subgroups

5

Remediation

Agentic loop: localize → fix → verify

6

Explainability

How different methods affect explainability

Static Analysis with SonarQube

What SonarQube Measures

Code Smells

Maintainability issues — overly complex methods, long parameter lists, duplicated blocks

Bugs

Reliability issues — null pointer dereferences, resource leaks, incorrect comparisons

Cognitive Complexity

Measure of how difficult code is to understand, beyond cyclomatic complexity

Duplications

Repeated code blocks indicating missing abstraction

Interpretation Tips

- Don't treat tool output at face value — context matters
- Compare relative to human-written baseline, not absolute thresholds
- LLM code often has more smells but fewer null-reference bugs
- Severity rating (Blocker → Info) guides prioritization
- False positive rate varies by language and project type

Hands-On Exercise 1 · Maintainability Analysis

```
from pipeline import run_sonarqube

results = run_sonarqube(
    code_dir='data/snippets/',
    sources=['llm_gpt4', 'human'],
    metrics=['smells','bugs','complexity']
)

plot_comparison(results)
```

Overview of Tasks

- [ ] Open Notebook
- [ ] Run SonarQube on the provided dataset
- [ ] Extract code smell counts per source
- [ ] Compare LLM vs. human distributions
- [ ] Generate comparison bar chart
- [ ] Note surprising findings

💬 *Discussion: Does higher code smell density in LLM code always mean lower quality? When might smells be misleading?*

Security Vulnerabilities in LLM-Generated Code

Key Finding: LLM-generated code often appears correct syntactically, yet harbors exploitable vulnerabilities that escape casual review.

CWE-78

OS Command Injection

subprocess with unsanitized input

CWE-89

SQL Injection

string-formatted SQL queries

CWE-22

Path Traversal

open() with user-controlled paths

CWE-798

Hardcoded Credentials

API keys embedded in source

CWE-502

Insecure Deserialization

pickle.loads() on untrusted data

CWE-330

Weak Randomness

random.random() for tokens

Bandit: Python Security Analysis

Severity × Confidence Matrix

	LOW conf.	MED conf.	HIGH conf.
HIGH sev.	Medium	High	**Critical** ⚠️
MED sev.	Low	Medium	High
LOW sev.	Informational	Low	Medium

Classification Workflow

- Run: bandit -r ./snippets -f json -o report.json
- Filter by severity × confidence ⟶ prioritize Critical/High
- Review flagged lines ⟶ distinguish true from false positives
- Aggregate counts per code source for comparison

Bandit output (JSON snippet)


```
{
     "code": "24     \"\"\"Return normal vector for first
three vertices.\"\"\"\n25     assert len(poly) >= 3\n26
x, y, z = poly[:3]\n",
     "col_offset": 4,
     "end_col_offset": 25,
     "filename": "src/cgpt_cont/interview/solution_1619.py",
     "issue_confidence": "HIGH",
     "issue_cwe": {
       "id": 703,
       "link":
"https://cwe.mitre.org/data/definitions/703.html"
     },
     "issue_severity": "LOW",
     "issue_text": "Use of assert detected. The enclosed code
will be removed when compiling to optimised byte code.",
     "line_number": 25,
     "line_range": [
       25
     ],
     "more_info":
"https://bandit.readthedocs.io/en/1.8.4.dev21/plugins/b101_as
sert_used.html",
     "test_id": "B101",
     "test_name": "assert_used"
   },
```

Hands-On Exercise 2 · Security Vulnerability Analysis

Example Findings

Source	Vuln ID	Severity
LLM-GPT4	B602	HIGH
Human	B605	MED
LLM-GPT4	B324	HIGH
LLM-Claude	B501	MED
Human	B108	LOW

Your goal: distinguish true positives from false positives and examine differences across code sources.

Overview of Tasks

- [ ] Open Notebook
- [ ] Run Bandit on the full dataset
- [ ] Filter HIGH severity findings
- [ ] Identify top 3 vulnerability types
- [ ] Compare LLM vs. human profiles
- [ ] Flag at least one false positive

💬 *Discussion: Are vulnerability rates higher in LLM code? Does the vulnerability type differ? What might explain this?*

Integrated Analysis

Combining Quality + Security into a Single Analysis View

Empirical Findings from Prior Studies

Maintainability

LLM code has 20–40% more code smells on average, with higher cognitive complexity per function (Molison et al., 2025)

Bugs

LLM code shows fewer null-reference bugs but more logical-flow defects that escape static detection (Liu et al., 2023)

Security

LLMs generate more injection-prone code; human code clusters around CWE-330 weak randomness (Siddiq et al., 2024)

Bias-Aware Evaluation

Why Bias Matters in AI Code Tools

- LLMs may generate code that reflects biases in training data (e.g., variable naming conventions, documentation style).
- Code quality scores may differ across subgroups defined by programming style, task domain, or model origin.
- Fairness-unaware pipelines can systematically penalize code from underrepresented sources.
- Intersectional bias: combinations of subgroup attributes amplify disparities beyond single-axis analysis.

Hands-On: Bias Exercise

1. Load a pre trained code model and define technical bias axes (e.g., cloud platform, programming language, frontend framework).
2. Measure baseline technology preferences using masked code/software engineering prompts.
3. Build and apply technical-bias projection subspaces to compare baseline and debiased model behavior.
4. Evaluate intersectional preferences across complete technology stacks (e.g., AWS × Python × React).
5. Quantify intersectional amplification and analyze how projection strength changes bias concentration.

Based on: Abule & Melo — 'Improving Fairness by Debiasing Intersectional Biases in Contextual AI Models' (CASCON 2025)

Let’s Get Started!

1) Scan the QR Code or visit the link below ↓

bit.ly/FSE-Intersectional-Bias

(link is case sensitive and requires Google Colab extension installed on Google account. If the link does not work, please access them via the google drive located at https://shorturl.at/0FvXX)

2) Make a copy of the notebook

3) Standby for further instructions! We are going to walk through the tasks shortly!

Short Break

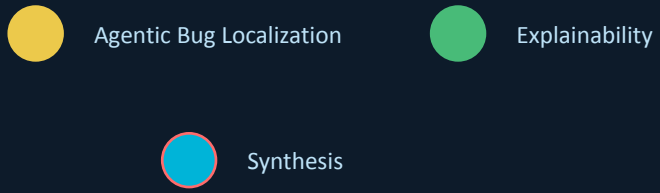
Part 2 starts at 11:00am

Stretch · Refill · Reflect on Part 1 findings

PART 2

Agentic Bug Localization, Explainability & Fairness

90 minutes · Advanced Topics + Hands-On



Agentic Bug Localization - Overview

Information Retrieval and Reformulation

- Learn to use a basic information retrieval pipeline
- Experiment with bug report reformulation
- Understand why reformulation can help retrieval

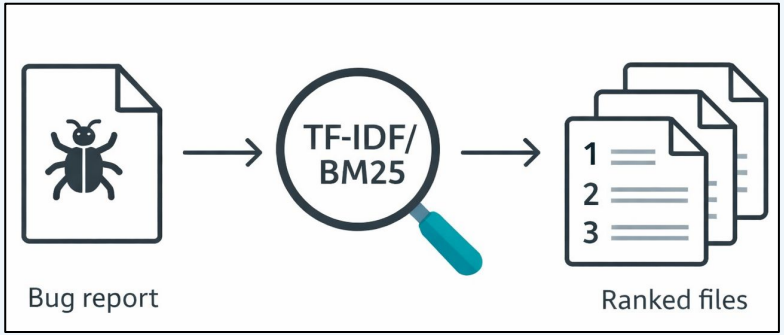
Agentic Bug Localization

- Analyze a simple agentic workflow for bug localization
- Inspect an agent trajectory to identify tool calls, decisions, errors, and recovery steps.
- Observe context growth, reflect on context management strategies.

From Traditional to Agentic Bug Localization

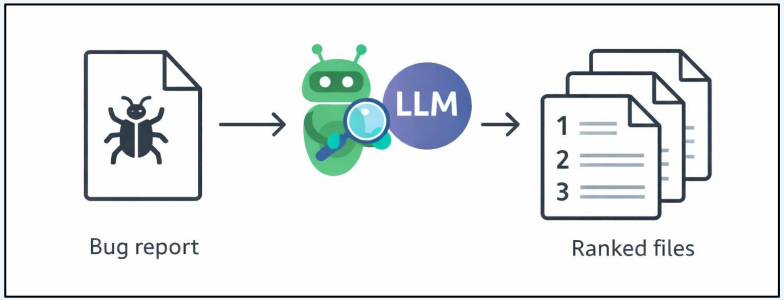
Then

- Matching bug reports to source code
- **Find buggy files for developers to inspect and fix**

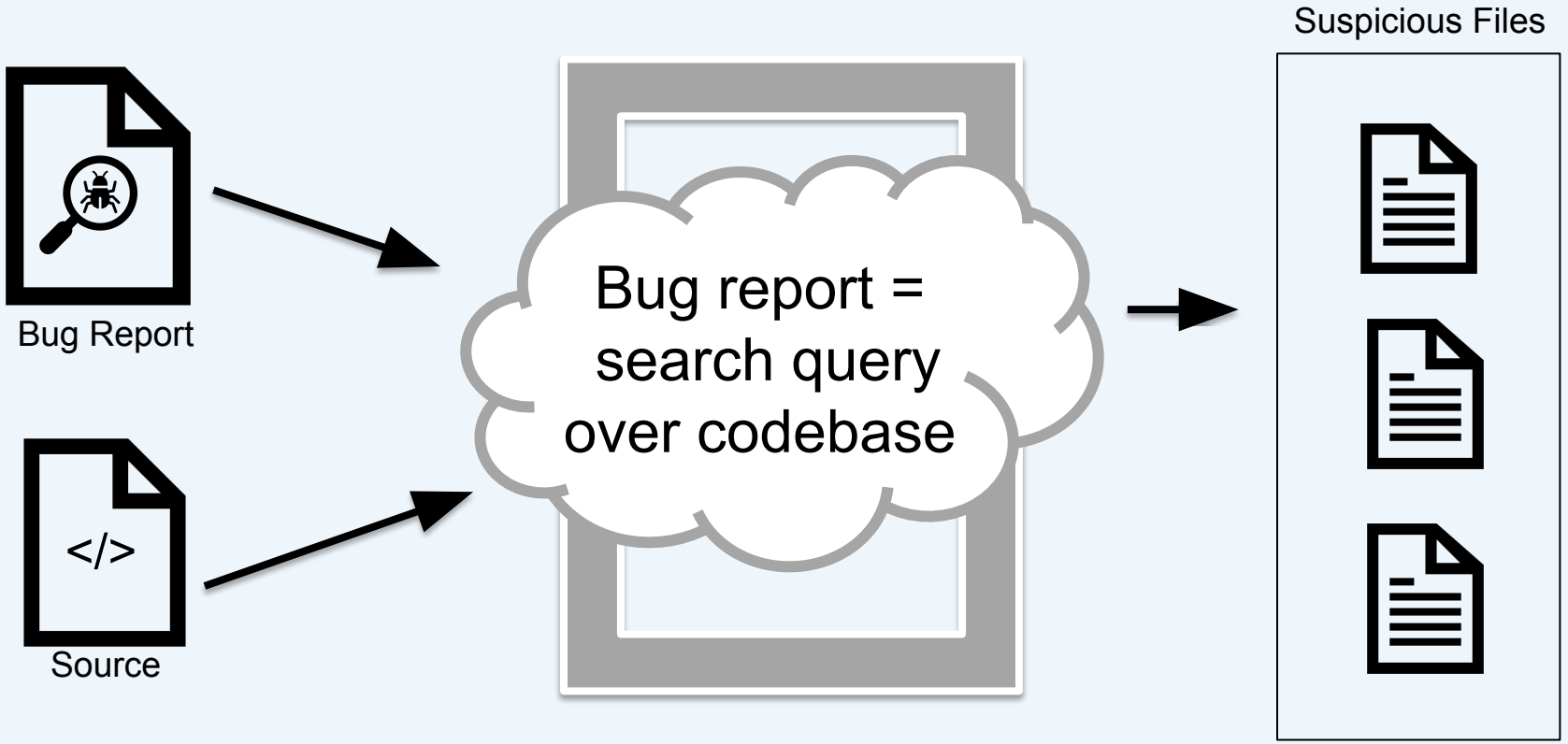


Now

- Traditional IR still used to reduce the search space
- **Find buggy files for LLMs to inspect and fix**



Overview - IRBL Pipeline



Based on: Caumartin & Melo — 'Reformulate, Retrieve, Localize: Agents for Repository-Level Bug Localization' (BoatSE @ ICSE 2026)

A Sample Bug Report

Bug Id: 4701

Instructions for reporter!

<!--
Welcome to the docker-compose issue tracker! Before creating an issue, please read the following:

1. This tracker should only be used to report bugs and request features / enhancements to docker-compose
(…)
-->

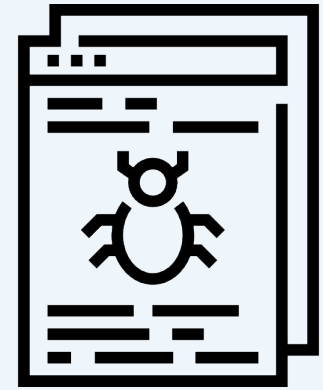
Boilerplate headers!

Description of the issue

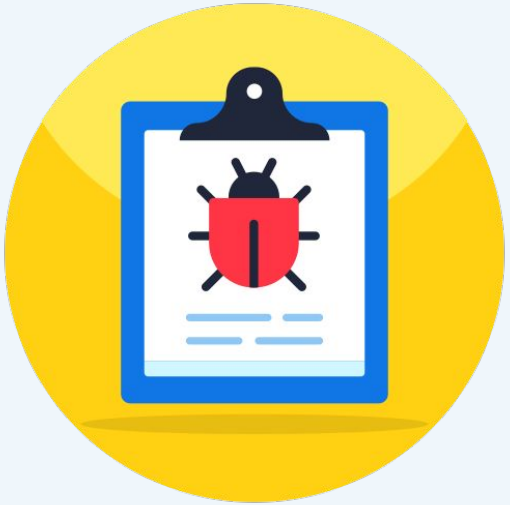
This is an usability issue. After https://github.com/docker/compose/pull/8082, running `docker-compose logs` will quit with `ERROR: configured logging driver does not support reading`, if any of the services has `logging.driver: none`. I would expect (…)

Bug Reports Are Noisy!

What if we reformulate the bug report?



Noisy Bug Report



Extracted Keywords

Based on: Caumartin & Melo — 'Reformulate, Retrieve, Localize: Agents for Repository-Level Bug Localization' (BoatSE @ ICSE 2026)

Reformulating a Bug Report - Example

Bug Id: 4064

Expected result
`pipenv clean` exits successfully.
Virtualenv is created.

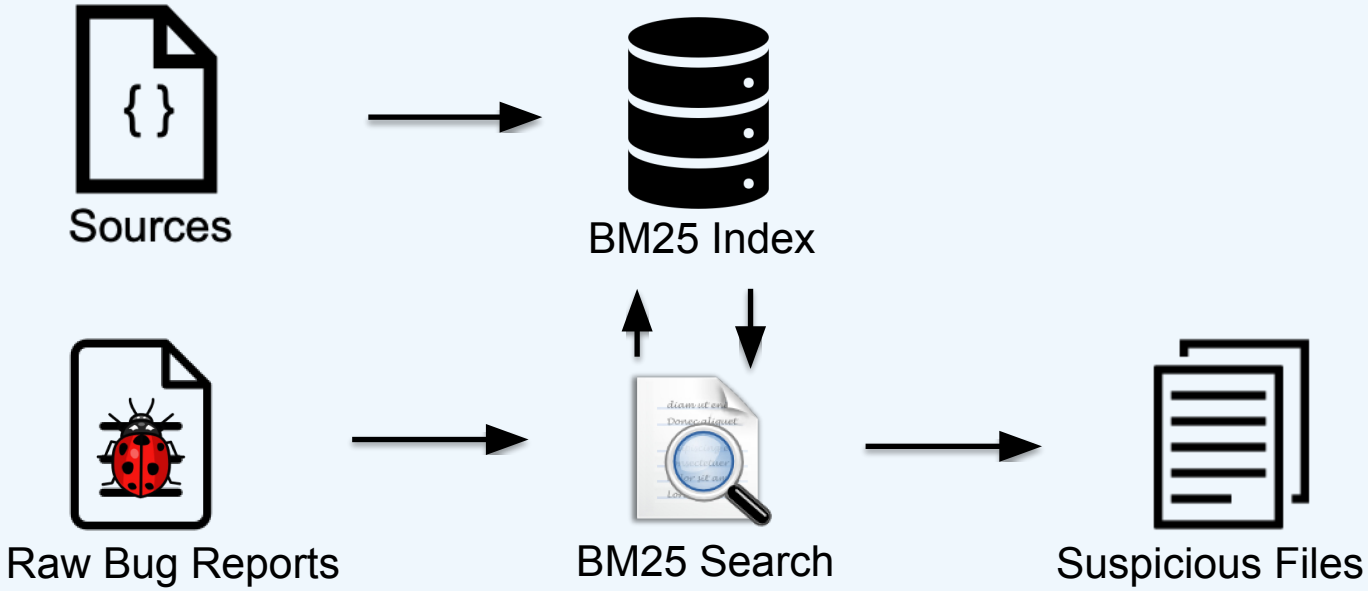
Actual result
$ pipenv clean
Creating a virtualenv for this project… [...]

Traceback (most recent call last):
in do_clean [...]
IndexError: list index out of range

Explanation	The bug occurs when running ‘pipenv clean’…
Path	src/
Filename	__init__.py
Identifiers	get_requirement, IndexError
Code snippet	Req = [r for r in requirements.parse(dep)[[0]]
Stack trace	Traceback … in do_clean […]
Error message	IndexError: list index out of range

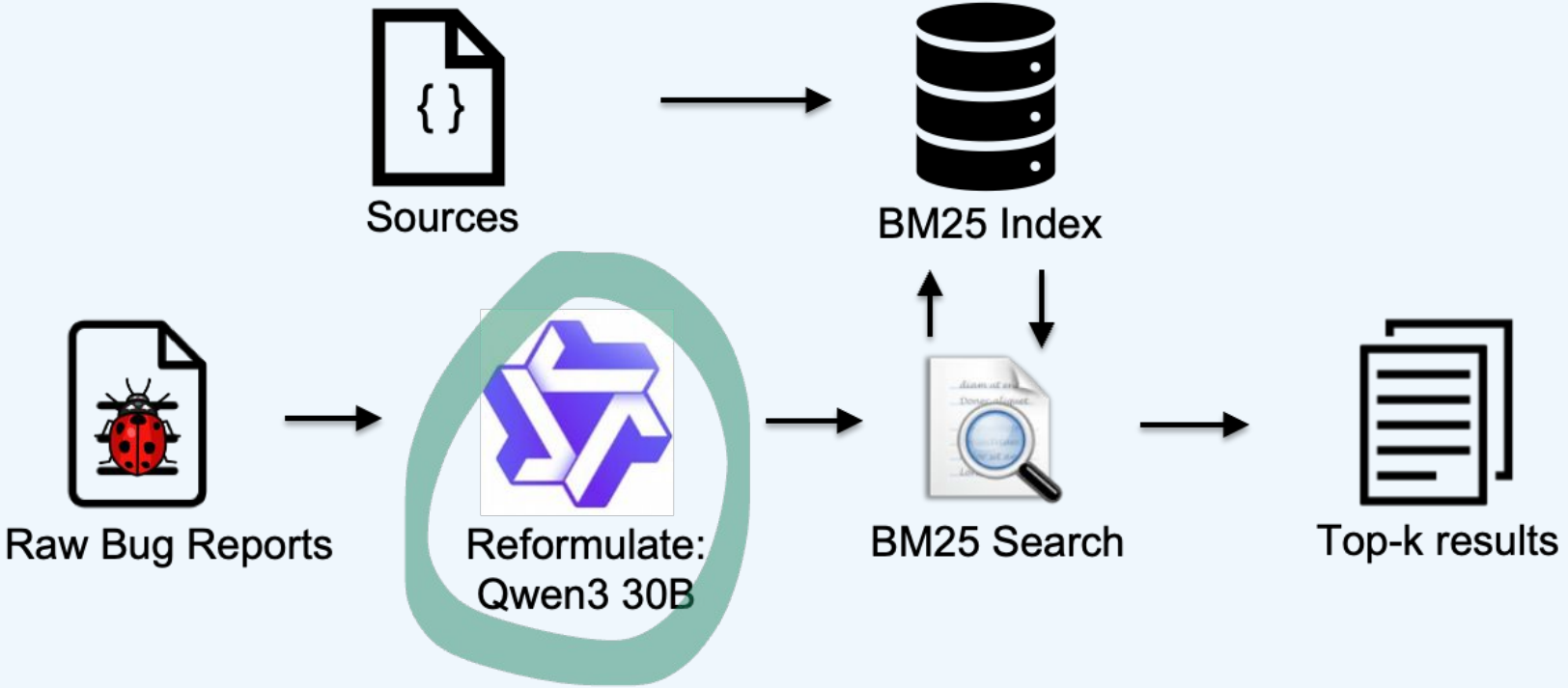
Based on: Caumartin & Melo — 'Reformulate, Retrieve, Localize: Agents for Repository-Level Bug Localization' (BoatSE @ ICSE 2026)

Simple retrieval workflow



Based on: Caumartin & Melo — 'Reformulate, Retrieve, Localize: Agents for Repository-Level Bug Localization' (BoatSE @ ICSE 2026)

Including a Reformulation Step



Based on: Caumartin & Melo — 'Reformulate, Retrieve, Localize: Agents for Repository-Level Bug Localization' (BoatSE @ ICSE 2026)

Hands-On Exercise: Information Retrieval and Reformulation

- Access the Colab notebook
- Run the IRBL pipeline on one instance of the SWE-bench Lite dataset

Access notebook

https://bit.ly/fse-tutorial-5

Information Retrieval Evaluation Metrics

Mean **A**verage **P**recision (MAP)
Mean of the average precision of several queries. Sensitive to where retrieved relevant files appear in the ranking. A relevant file at rank 1 contributes more to average precision than a relevant file at rank 10.

Hit@K
Measures if at least one ground truth files appears in the top-k retrieved files.

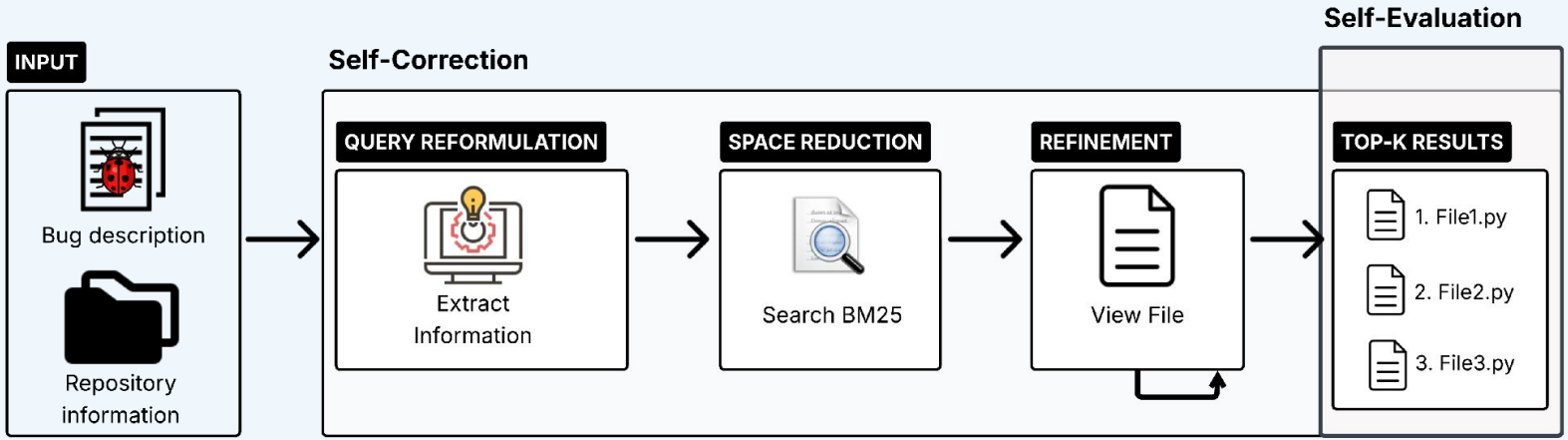
Experimental Results on Long Code Arena and SWE-bench Lite

Dataset	BM25 Query	MAP@5	Hit@1	Hit@5
LCA	**Baseline**	0.250	0.320	0.653
	Identifiers, code snippets	0.336	**0.447** +28%	0.640
	Explanation, identifiers, code snippets	**0.339** +26%	0.440	**0.733** +11%
SWE	**Baseline**	0.488	0.400	0.640
	Identifiers, code snippets	0.538	0.470	0.640
	Explanation, identifiers, code snippets	**0.584** +16%	**0.476** +16%	**0.757** +16%

Based on: Caumartin & Melo — 'Reformulate, Retrieve, Localize: Agents for Repository-Level Bug Localization' (BoatSE @ ICSE 2026)

Agentic Bug Localization

Overview of our Agentic Workflow



Based on: Caumartin & Melo — 'Reformulate, Retrieve, Localize: Agents for Repository-Level Bug Localization' (BoatSE @ ICSE 2026)

Experimental Results on SWE-bench Lite

Model	MAP@5	Hit@1	Hit@5
Best BM25 config	0.584	0.476	0.757
Qwen 2.5 Coder 32B	0.746	0.648	**0.888**
Qwen 3 30B	**0.784** +26%	**0.674** +27%	**0.888** +15%

Based on: Caumartin & Melo — 'Reformulate, Retrieve, Localize: Agents for Repository-Level Bug Localization' (BoatSE @ ICSE 2026)

Agentic Bug Localization

Focus of this part of the tutorial

- **Debug** an agent trajectory
- Observe context growth and reflect on **context management** strategies.

Context management, in a nutshell

- Managing information sent to a LLM so it stays relevant and as concise as possible
- LLMs are sensitive to context overload and lose accuracy with too much information

Hands-On Exercise · Agentic Localization Loop

- Access the Colab notebook

- Run the **agentic loop** on one instance of the SWE-bench Lite dataset

The reformulation and agentic loop source code is available here:
https://github.com/gencau/boatse-extractor

Access notebook

https://bit.ly/fse-tutorial-5

Hands-On Exercise · Trajectory Review

- LLMs are prone to **data leakage**. Take a closer look at the result of the *extract_relevant* tool call. What do you notice?

- Did the agent make any **errors** in your trajectory? Was it able to fix it?

- **Notice!** In the last steps, the agent runs a self-validation step, where it evaluates its response by ranking each file from 1 to 10.

Hands-On Exercise · Context Management

- **Context accumulates** as the conversation grows. Check how the “prompt” message type grows as the conversation progresses.

- **Final self review** starts a new conversation from scratch, using essential information to assess the result: issue description, tool results, final response.

What context management strategies would you use in a long conversation?

https://bit.ly/menti-5-1

Takeaways

- **Reformulation is a context management strategy**
  Removes noise, focuses on relevant signals
- **Agent trajectories are observable evidence**
  Tool calls expose errors, retries, and prompt content
- **Context size can grow quickly**
  Agentic workflows need explicit strategies to compress, reset, and validate context

Thank you!

More information about context management
https://claude-code.program.repair/

Our paper Reformulate, Retrieve, Localize: Agents for Bug Localization

https://bit.ly/3SHrZbS

Explainability for Developer Trust

A generated patch is not enough. Developers need to understand whether it is correct, why it works, and whether they would trust it enough to act.

Participants will:

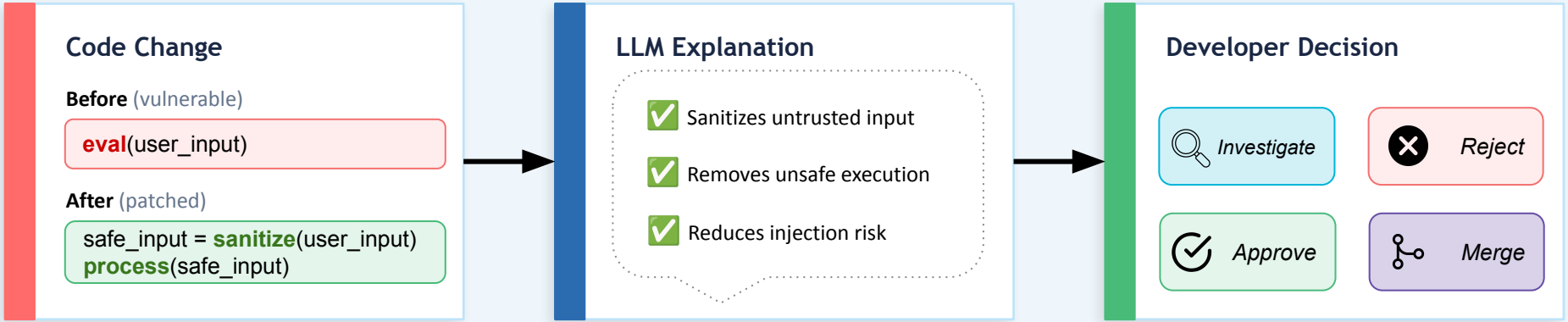
- Compare LLM-generated explanations for code changes and remediations
- Evaluate explanation quality
- Reflect on how explanations influence trust in AI-assisted development

4

TRUST

Will a developer actually accept the fix?

Part 2 · Corey
Explainability for developer trust



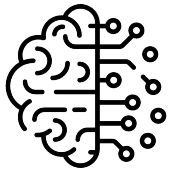
The Developer’s Dilemma

Static Analysis Tools

Detects vulnerability patterns, but often missing project-specific context.

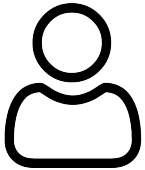
Large Language Models

Can generate candidate repairs, but fixes are not automatically validated.



Developer

Must assess the vulnerability, patch, rationale, and risk before approving a change.



A patch can look plausible, pass a tool, and still be hard to approve.

Hands-On Exercise · What Would You Merge?

```python
import os

def execute_command(user_input):
    command = user_input
    os.system(command)

user_input = input("Enter your command: ")
execute_command(user_input)
```

Hands-On Exercise · What Would You Merge?

```python
import os

def execute_command(user_input):
    command = user_input
    os.system(command)

user_input = input("Enter your command: ")
execute_command(user_input)
```

```python
import shlex
import subprocess
import sys

ALLOWED_COMMANDS = {'echo', 'pwd', 'date', 'ls', 'whoami', 'uname', 'hostname',
    'id', 'uptime', 'df', 'du', 'sleep'}

def execute_command(user_input):
    tokens = shlex.split(user_input)
    if not tokens:
        return
    cmd = tokens[0]
    if cmd not in ALLOWED_COMMANDS:
        print("Error: Command not allowed.")
        return
    try:
        subprocess.run(tokens, check=True)
    except subprocess.CalledProcessError as e:
        print(f"Command failed with exit code {e.returncode}", file=sys.stderr)
    except Exception as e:
        print(f"Error executing command: {e}", file=sys.stderr)

if __name__ == '__main__':
    user_input = input("Enter your command: ")
    execute_command(user_input)
```

Whitelist

Safer

```python
import shlex
import subprocess
import sys

ALLOWED_COMMANDS = {'echo', 'pwd', 'date', 'ls', 'whoami', 'uname', 'hostname',
    'id', 'uptime', 'df', 'du', 'sleep'}

def execute_command(user_input):
    tokens = shlex.split(user_input)
    if not tokens:
        return
    cmd = tokens[0]
    if cmd not in ALLOWED_COMMANDS:
        print("Error: Command not allowed.")
        return
    try:
        subprocess.run(tokens, check=True)
    except subprocess.CalledProcessError as e:
        print(f"Command failed with exit code {e.returncode}", file=sys.stderr)
    except Exception as e:
        print(f"Error executing command: {e}", file=sys.stderr)

if __name__ == '__main__':
    user_input = input("Enter your command: ")
    execute_command(user_input)
```

Would you merge in this patch?

Why or why not?

Hands-On Exercise · What Would You Merge? - Discussion

Q1

What about the patch makes you hesitant to merge it?

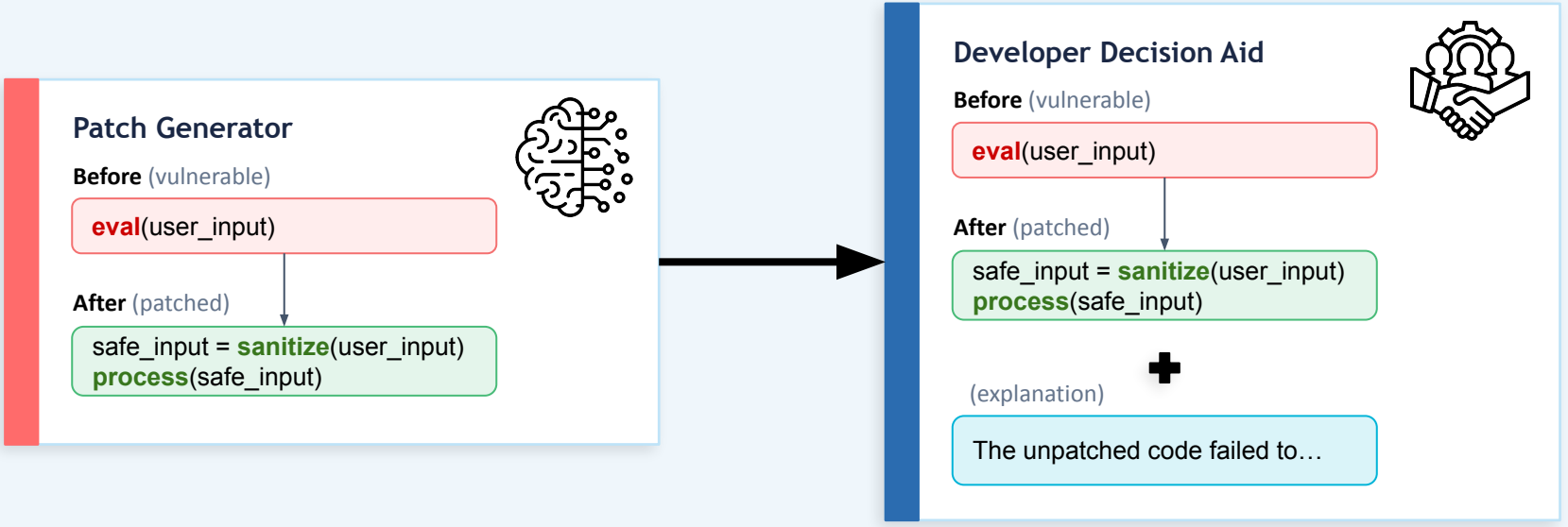
- Missing context
- Unfamiliar with the code/language
- Not a security researcher

Q2

What extra information would you need before feeling confident in approving the patch?

- Surrounding patch context
- Information about libraries used
- Info about the vulnerability

AVR as Review Support

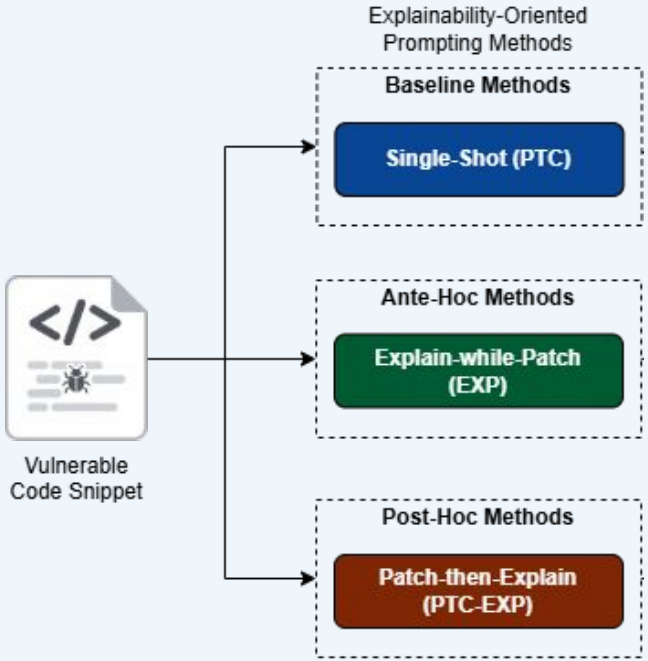


The goal is not only to generate a fix, but to **help a developer confidently accept it.**

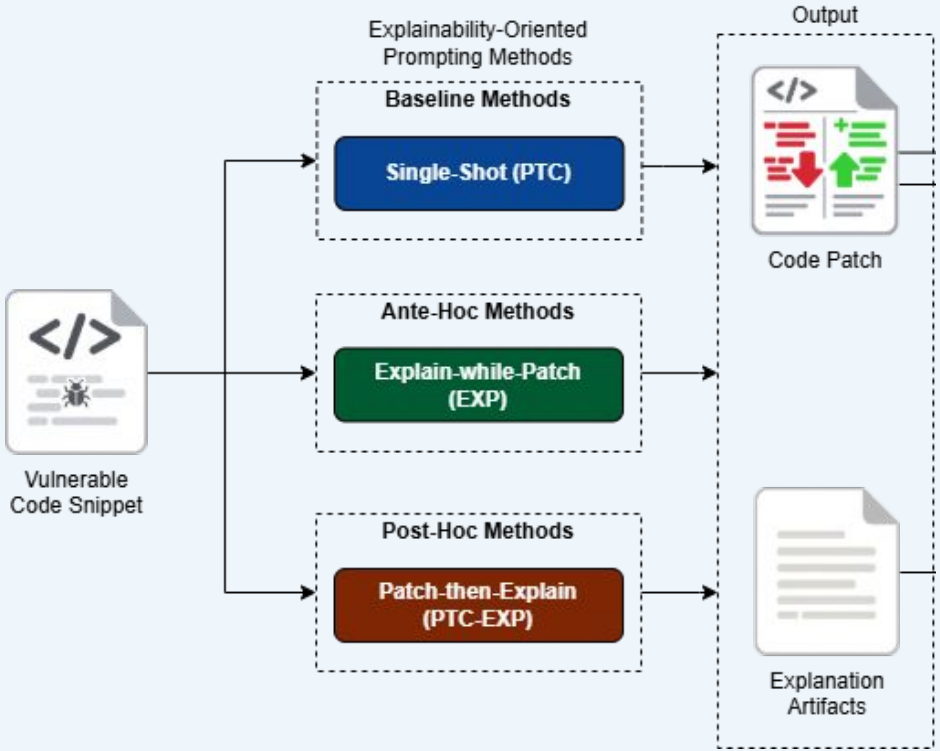
Simplified Methodology

Vulnerable
Code Snippet

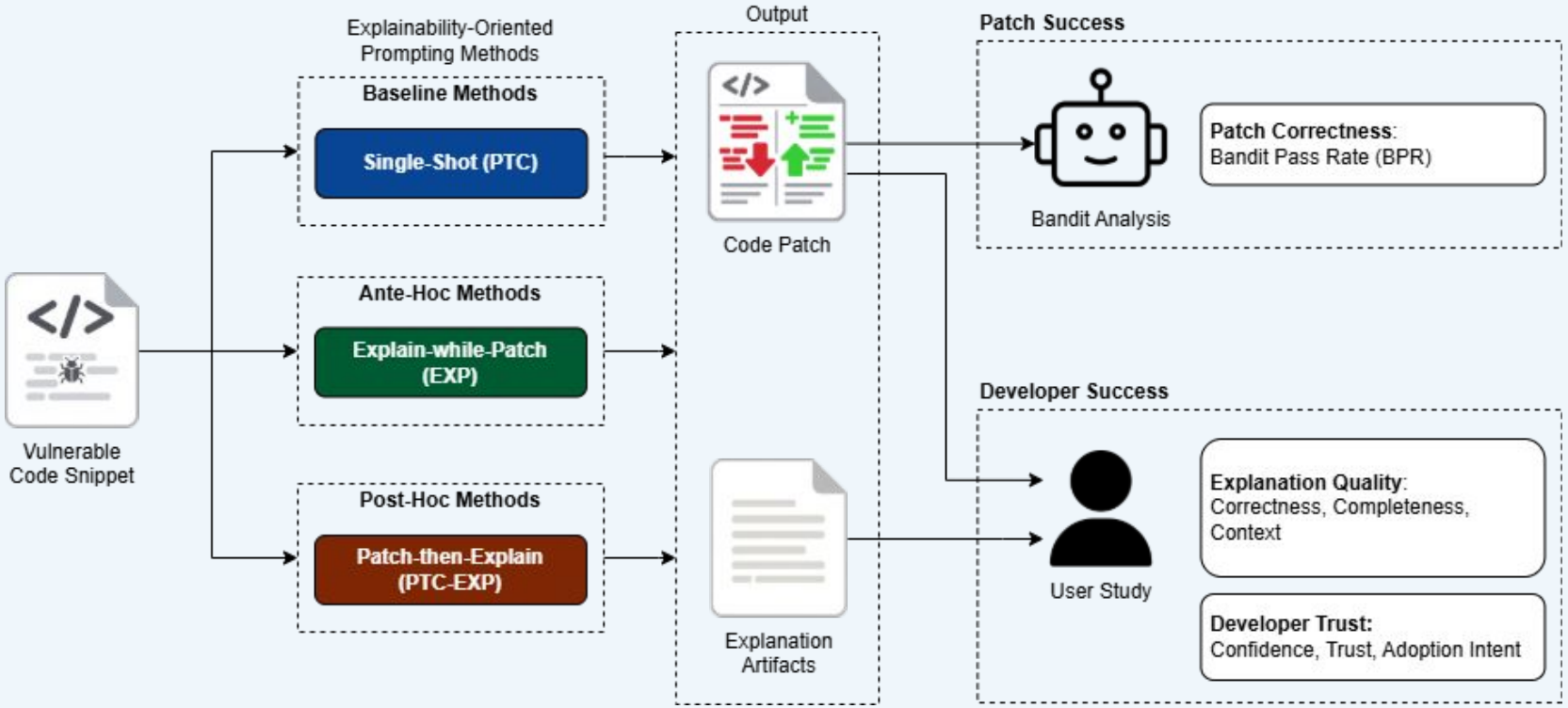
Simplified Methodology



Simplified Methodology



Simplified Methodology



Simplified Methodology - Methods Explanation

Baseline

Single-Shot (PTC)

Patched Code

Ante-Hoc

Explain-while-Patch (EXP)

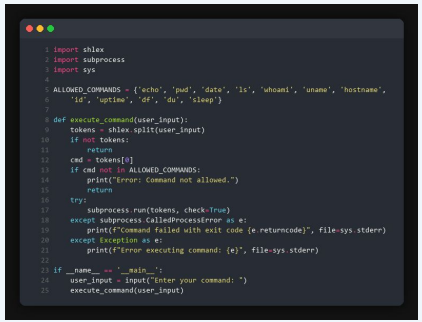
Patched Code + Explanation

Post-Hoc

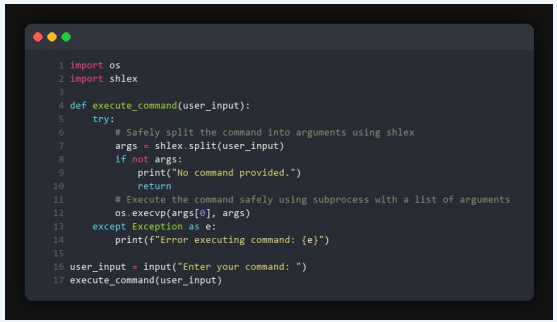
Patch-then-Explain (PTC-EXP)

Patched Code

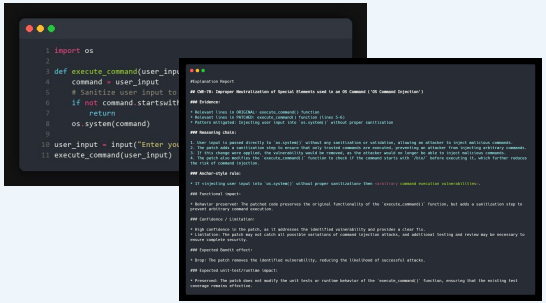
Explanation



```python
import shlex
import subprocess
import sys

ALLOWED_COMMANDS = {'echo', 'pwd', 'date', 'ls', 'whoami', 'uname', 'hostname',
    'id', 'uptime', 'df', 'du', 'sleep'}

def execute_command(user_input):
    tokens = shlex.split(user_input)
    if not tokens:
        return
    cmd = tokens[0]
    if cmd not in ALLOWED_COMMANDS:
        print("Error: Command not allowed.")
        return
    try:
        subprocess.run(tokens, check=True)
    except subprocess.CalledProcessError as e:
        print(f"Command failed with exit code {e.returncode}", file=sys.stderr)
    except Exception as e:
        print(f"Error executing command: {e}", file=sys.stderr)

if __name__ == '__main__':
    user_input = input("Enter your command: ")
    execute_command(user_input)
```



```python
import os
import shlex

def execute_command(user_input):
    try:
        # Safely split the command into arguments using shlex
        args = shlex.split(user_input)
        if not args:
            print("No command provided.")
            return
        # Execute the command safely using subprocess with a list of arguments
        os.execvp(args[0], args)
    except Exception as e:
        print(f"Error executing command: {e}")

user_input = input("Enter your command: ")
execute_command(user_input)
```



```python
import os

def execute_command(user_inp
    command = user_input
    # Sanitize user input to
    if not command.startswit
        return
    os.system(command)

user_input = input("Enter yo
execute_command(user_input)
```

```
#Explanation Report

## CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

### Evidence:

* Relevant lines in ORIGINAL `execute_command()` function
* Relevant lines in PATCHED `execute_command()` function (lines 5-6)
* Pattern mitigated: `Directly pass user input into 'os.system()' without proper sanitization`

### Reasoning chain:

1. User input is passed directly to `os.system()` without any sanitization or validation, allowing an attacker to inject malicious commands.
2. The patch adds a sanitization step to ensure that only trusted commands are executed, preventing an attacker from injecting arbitrary commands.
3. If this change were applied, the vulnerability would be removed, as the attacker would no longer be able to inject malicious commands.
4. The patch introduces a new function `execute_command()` that checks if the command starts with `ls -la` before executing it, which further reduces the risk of command injection.

### Anchor-style rule:

* If `<injecting user input into 'os.system()' without proper sanitization>` then `<arbitrary command execution vulnerabilities>`.

### Functional impact:

* Behavior preserved: The patched code preserves the original functionality of the `execute_command()` function, but adds a sanitization step to prevent arbitrary command execution.

### Confidence / Limitation:

* High confidence in the patch, as it addresses the identified vulnerability and provides a clear fix.
* Limitation: The patch may not catch all possible variations of command injection attacks, and additional testing and review may be necessary to ensure complete security.

### Expected Benefit effort:

* Drop: The patch removes the identified vulnerability, reducing the likelihood of successful attacks.

### Expected unit-test/runtime impact:

* Preserved: The patch does not modify the unit tests or runtime behavior of the `execute_command()` function, ensuring that the existing test coverage remains effective.
```

Simplified Methodology - Measurement

Patch Success Metrics

1. **Net Improvement Rate (NIR):** Captures the proportion of generated patches that **reduce Bandit security findings**; how often the model improves security

$$NIR = \frac{\{\text{patches where Bandit findings decrease}\}}{\{\text{generated patches evaluated}\}}$$

Developer Success Metrics

1. **Correctness:** Does the explanation accurately describe what changes and why it fixes the vulnerability?
2. **Completeness:** Does it cover the root cause, impact, and rationale for the fix?
3. **Context:** Does it explain why this matters in a real security review?
4. **Developer Confidence:** After reviewing this change, how confident are you with merging or approving it?

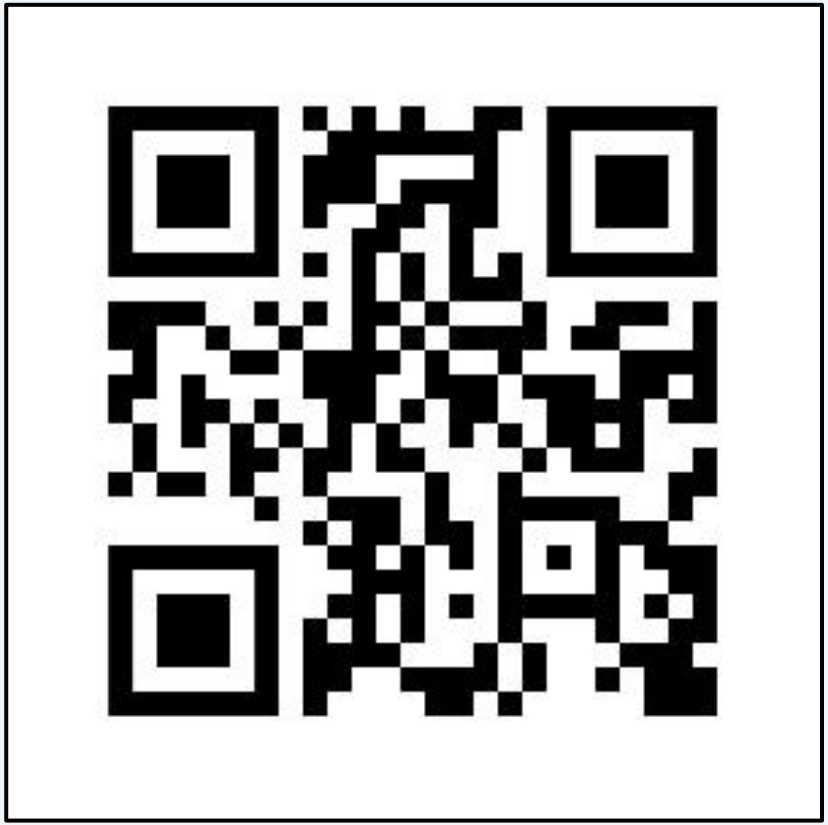
Source: From Anecdotal Evidence to Quantitative Evaluation Methods: A Systematic Review on Evaluating Explainable AI

Colab Walkthrough

Google Colab

- Walkthrough of Bandit detection
- One simple vulnerability example
- Our 3 explainability methods (baseline, ante-hoc, post-hoc)
- Comparison to our study findings

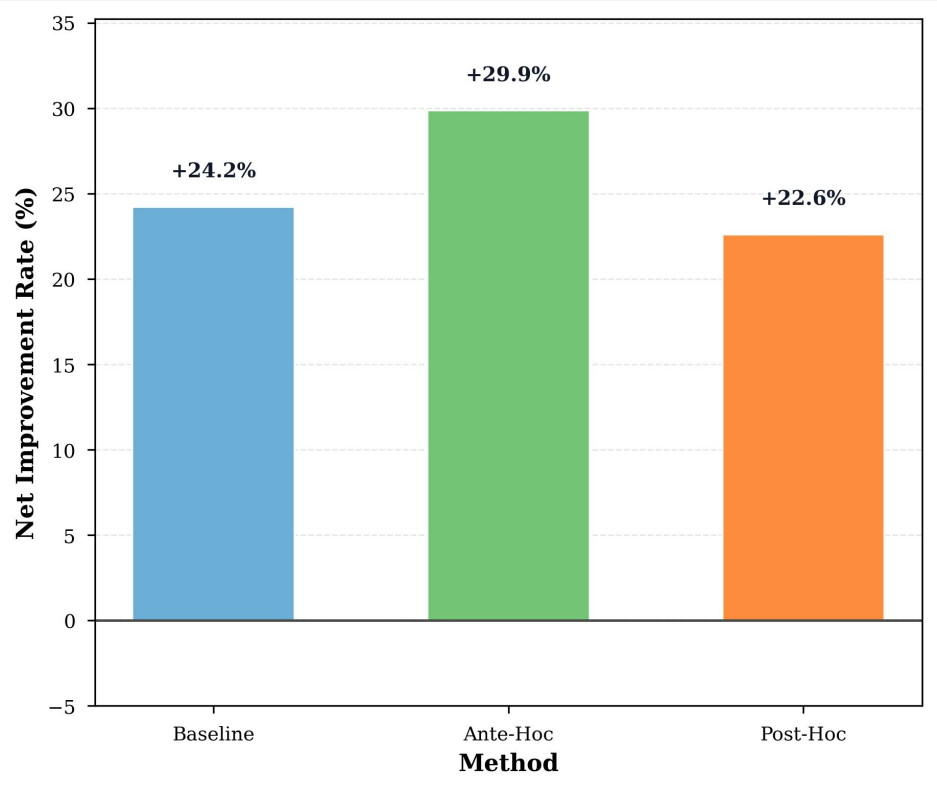
https://shorturl.at/VWABX



Explanations Can Improve Repairs

Takeaway: Ante-Hoc explanation can improve patch performance

Ante-hoc methods integrate explanation into patch generation, and can guide how a model generates the fix.



Hands-On Exercise · Developer Patch Preference

https://menti.com/

Code to Join

6157 6376

Developer Success Metrics

1. **Correctness:** Does the explanation accurately describe what changes and why it fixes the vulnerability?
2. **Completeness:** Does it cover the root cause, impact, and rationale for the fix?
3. **Context:** Does it explain why this matters in a real security review?
4. **Developer Confidence:** After reviewing this change, how confident are you with merging or approving it?

Hands-On Exercise · Developer Patch Preference

A

```python
import os
import shlex

def execute_command(user_input):
    try:
        # Safely split the command into arguments using shlex
        args = shlex.split(user_input)
        if not args:
            print("No command provided.")
            return
        # Execute the command safely using subprocess with a list of arguments
        os.execvp(args[0], args)
    except Exception as e:
        print(f"Error executing command: {e}")

user_input = input("Enter your command: ")
execute_command(user_input)
```

6157 6376

Hands-On Exercise · Developer Patch Preference

B

```python
import os

def execute_command(user_input):
    command = user_input
    # Sanitize user input to prevent arbitrary command execution
    if not command.startswith("/bin/"):
        return
    os.system(command)

user_input = input("Enter your command: ")
execute_command(user_input)
```

6157 6376

Hands-On Exercise · Developer Patch Preference

B

6157 6376

#Explanation Report

CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

Evidence:

* Relevant lines in ORIGINAL: execute_command() function
* Relevant lines in PATCHED: execute_command() function (lines 5-6)
* Pattern mitigated: Injecting user input into `os.system()` without proper sanitization

Reasoning chain:

1. User input is passed directly to `os.system()` without any sanitization or validation, allowing an attacker to inject malicious commands.
2. The patch adds a sanitization step to ensure that only trusted commands are executed, preventing an attacker from injecting arbitrary commands.
3. If this change were applied, the vulnerability would be removed, as the attacker would no longer be able to inject malicious commands.
4. The patch also modifies the `execute_command()` function to check if the command starts with `/bin/` before executing it, which further reduces the risk of command injection.

Anchor-style rule:

* If <injecting user input into `os.system()` without proper sanitization> then <arbitrary command execution vulnerabilities>.

Functional impact:

* Behavior preserved: The patched code preserves the original functionality of the `execute_command()` function, but adds a sanitization step to prevent arbitrary command execution.

Confidence / Limitation:

* High confidence in the patch, as it addresses the identified vulnerability and provides a clear fix.
* Limitation: The patch may not catch all possible variations of command injection attacks, and additional testing and review may be necessary to ensure complete security.

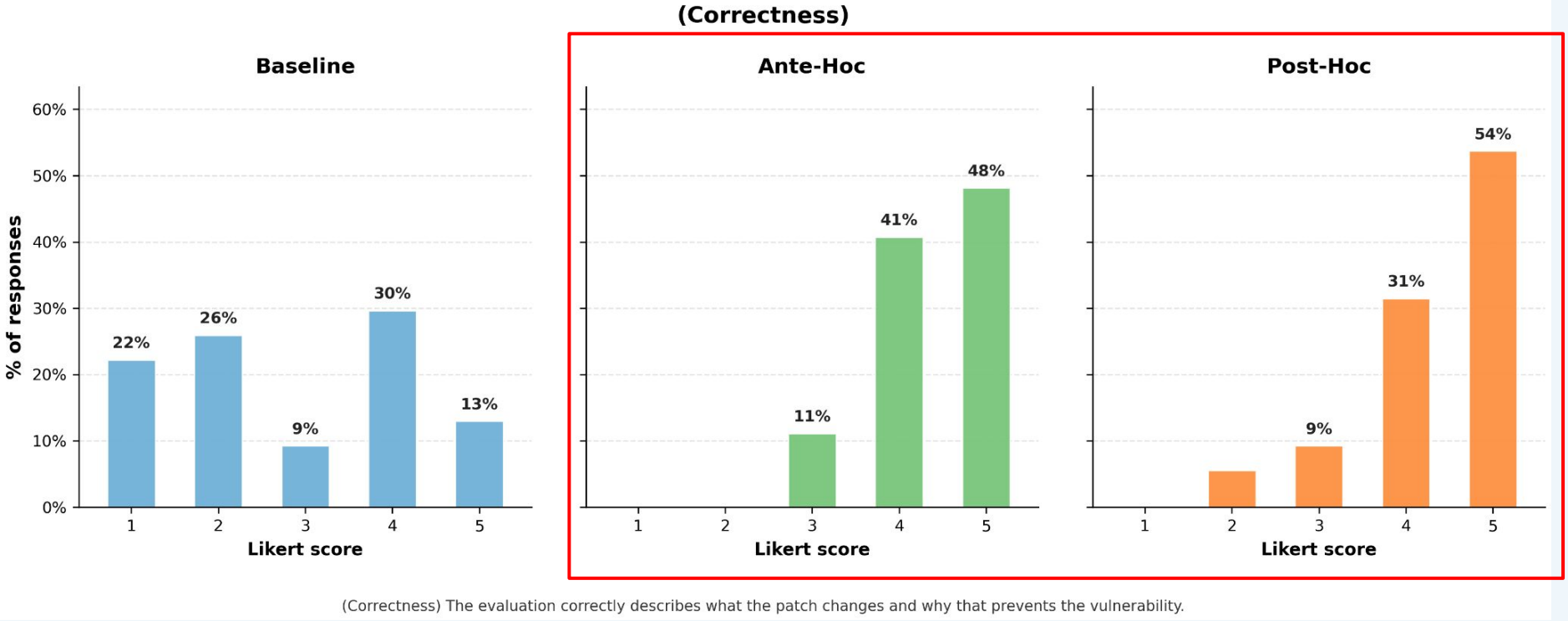
Expected Bandit effect:

* Drop: The patch removes the identified vulnerability, reducing the likelihood of successful attacks.

Expected unit-test/runtime impact:

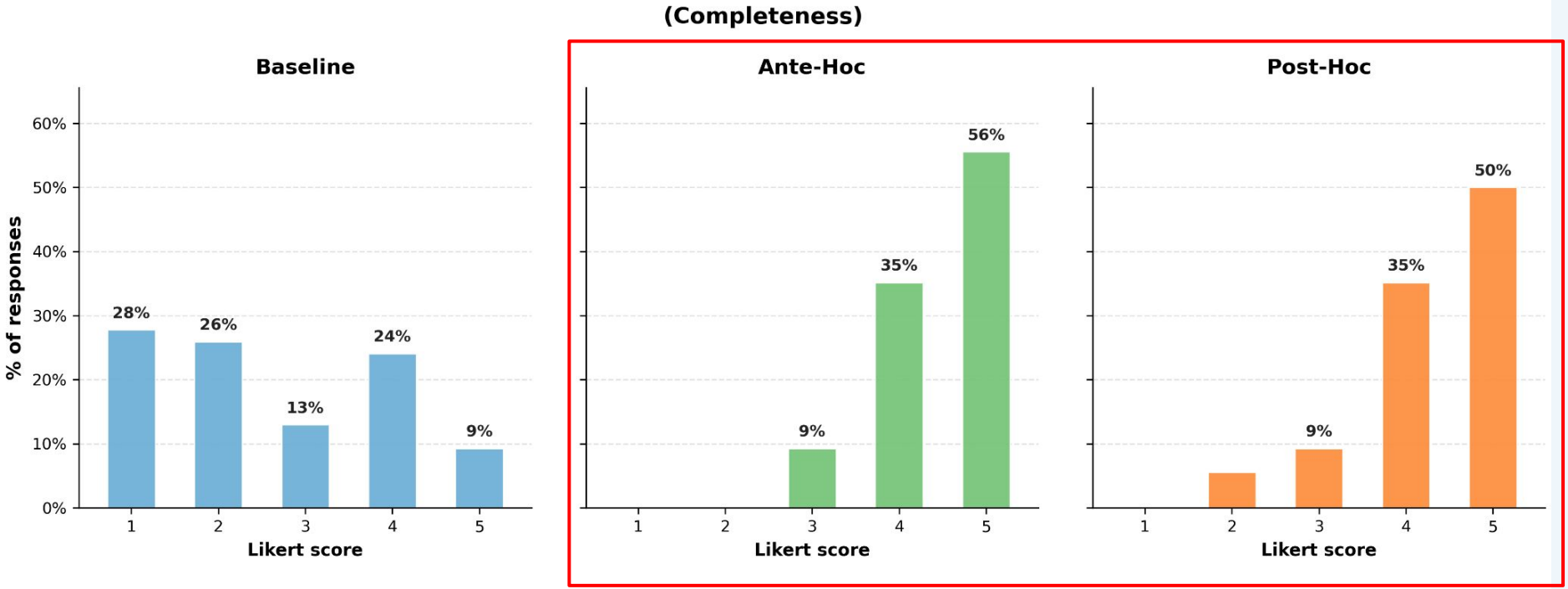
* Preserved: The patch does not modify the unit tests or runtime behavior of the `execute_command()` function, ensuring that the existing test coverage remains effective.

Correctness



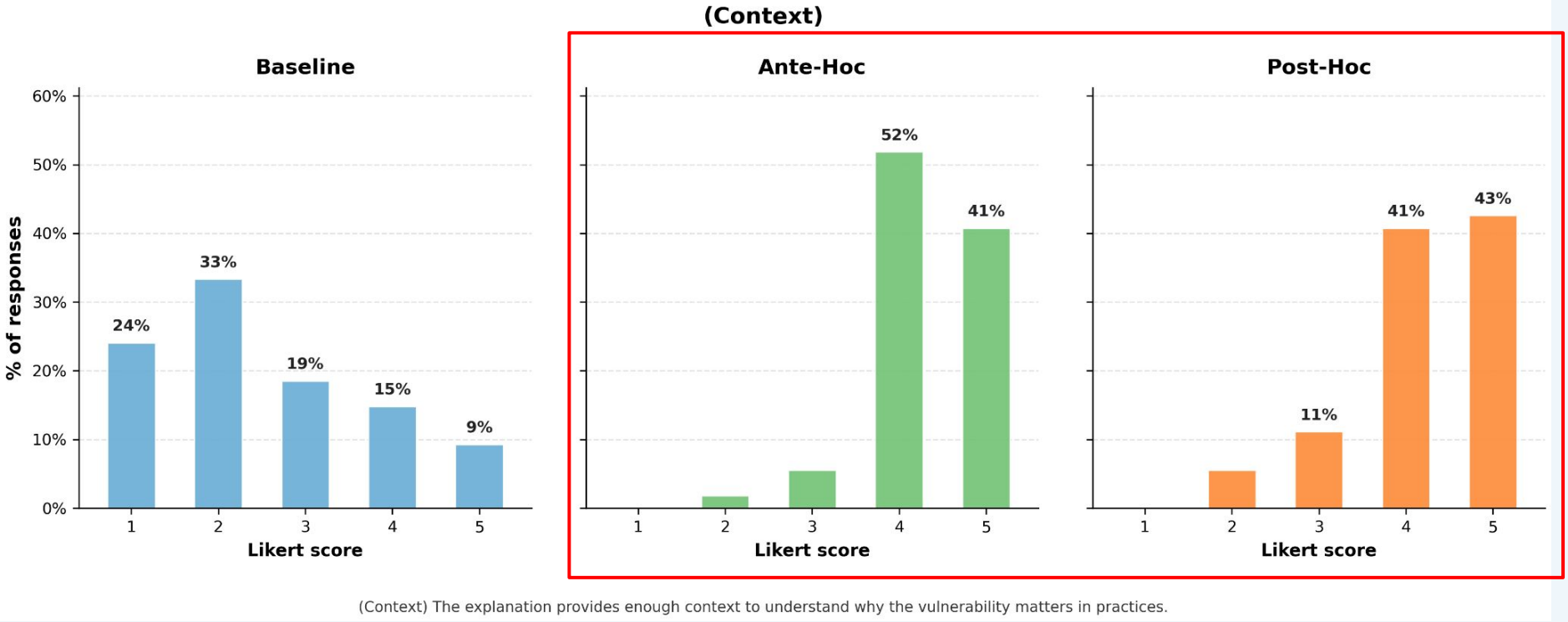
(Correctness) The evaluation correctly describes what the patch changes and why that prevents the vulnerability.

Completeness



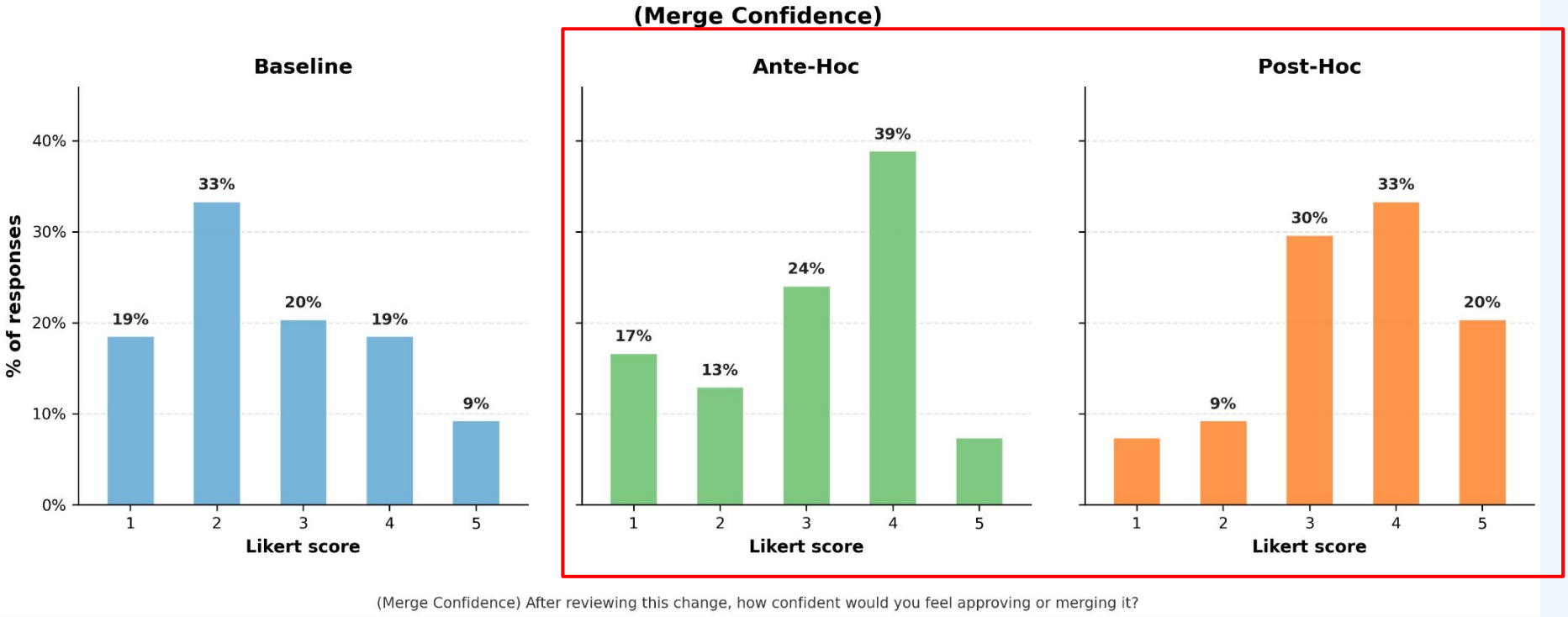
(Completeness) The explanation covers the key aspects of the vulnerability, including cause, impact, and fix.

Context

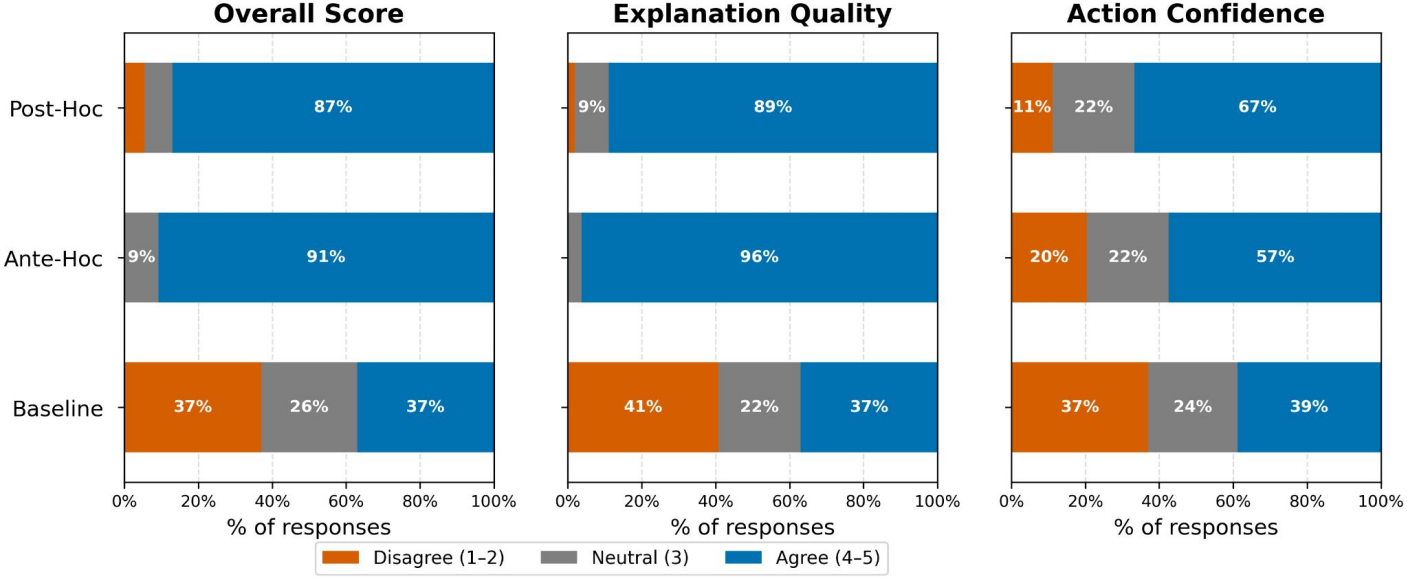


(Context) The explanation provides enough context to understand why the vulnerability matters in practices.

Merge Confidence



Explanation Quality is important but =/= Trust

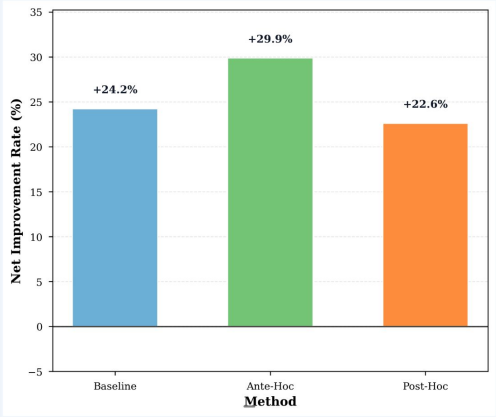


Takeaway: Explanation quality does not automatically create trust

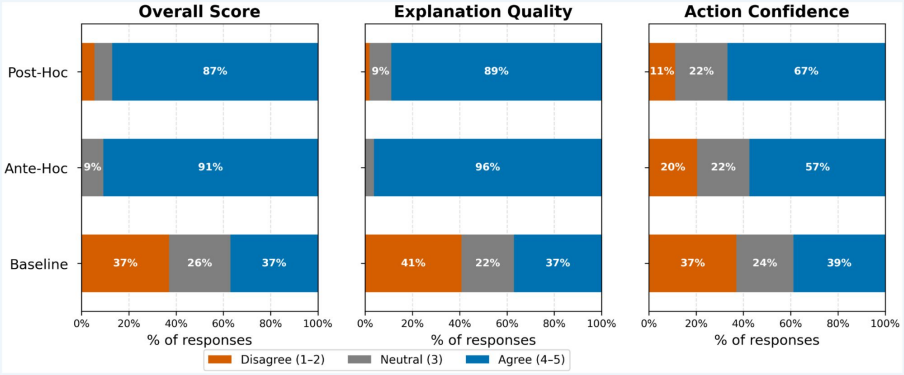
Explanation-bearing methods were rated more favorably than no-explanation baselines; however, high-quality explanations did not always translate into confidence to approve or merge the patch.

Explainability for Developer Trust

Explanations Can Improve Repairs



Explanation Quality is important but =/= Trust



Takeaway: Ante-Hoc explanation can improve patch performance

Ante-hoc methods integrate explanation into patch generation, and can guide how a model generates the fix.

Takeaway: Explanation quality does not automatically create trust

Explanation-bearing methods were rated more favorably than no-explanation baselines; however, high-quality explanations did not always translate into confidence to approve or merge the patch.

Synthesis

Human-in-the-Loop &

The Future of AI-Assisted SE

Human Oversight

Automated pipelines surface issues; developers make final judgments on risk vs. cost trade-offs

Trust Calibration

Explainability outputs should inform — not replace — developer review of AI-generated artifacts

Fairness by Design

Bias-aware evaluation must be integrated early, not retrofitted as an afterthought

Reproducibility

Reusable, versioned pipelines enable cumulative progress and defensible quality assessments

Special Issue

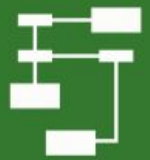
Crowdsourcing, Human-AI Interaction, and the Future of Digital Platforms

Guest Editors

Prof. Dr. Daniel Schneider
Dr. Glaucia Melo

Deadline

31 December 2026



algorithms

References

[1] Molison, A. S., Moraes, M., Melo, G., Santos, F., & Assuncao, F. (2025). Is LLM-Generated Code More Maintainable & Reliable Than Human-Written Code? ESEM 2025, pp. 151–162.

[2] Siddiq, M. L., Roney, L., Zhang, J., & Santos, J. C. D. S. (2024). Quality assessment of ChatGPT generated code and their use by developers. MSR 2024, pp. 152–156.

[3] Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of LLMs for code generation. NeurIPS 2023, vol. 36, pp. 21558–21572.

[4] Caumartin, G., & Melo, G. (2026). Reformulate, Retrieve, Localize: Agents for Repository-Level Bug Localization. BoatSE @ ICSE 2026.

[5] Abule, N., & Melo, G. (2025). Improving fairness by debiasing intersectional biases in contextual AI models. CASCON 2025.

Thank You!

Questions & Discussion

Glaucia Melo · glaucia@torontomu.ca

Jessica Pourleyli · jessica.pourleyli@torontomu.ca

Genevieve Caumartin · genevieve.caumartin@mail.concordia.ca

Corey Yang-Smith · corey.yangsmith@ucalgary.ca

Notebook & Materials

The Google Colab notebook, dataset, and all materials will be shared with participants. The pipeline is fully reusable — adapt it to your own research or production workflows.