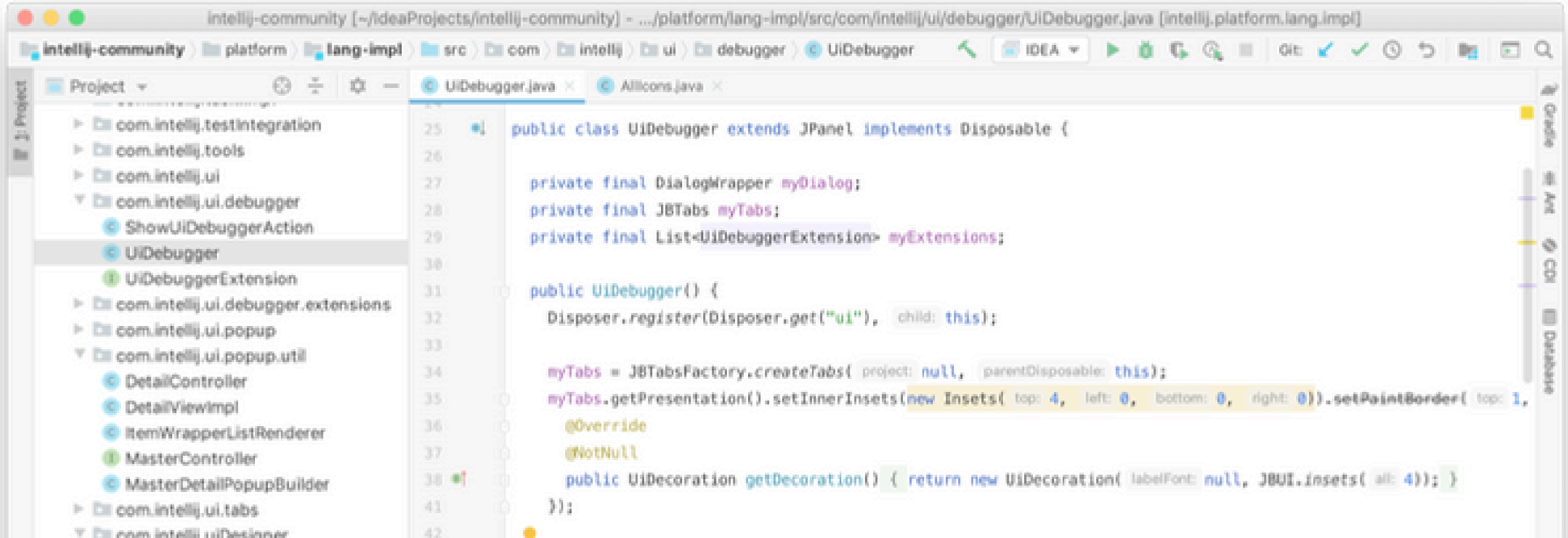
Benchmarking LLM-Generated Code Security via Static Analysis and CWE Scanning

Aadvait Hirde

Kelley Data Science & AI Lab



Outline

- Problem Introduction
- Research Objective
- Literature Review
- Gaps and Questions
- Research Design
- Timeline
- References

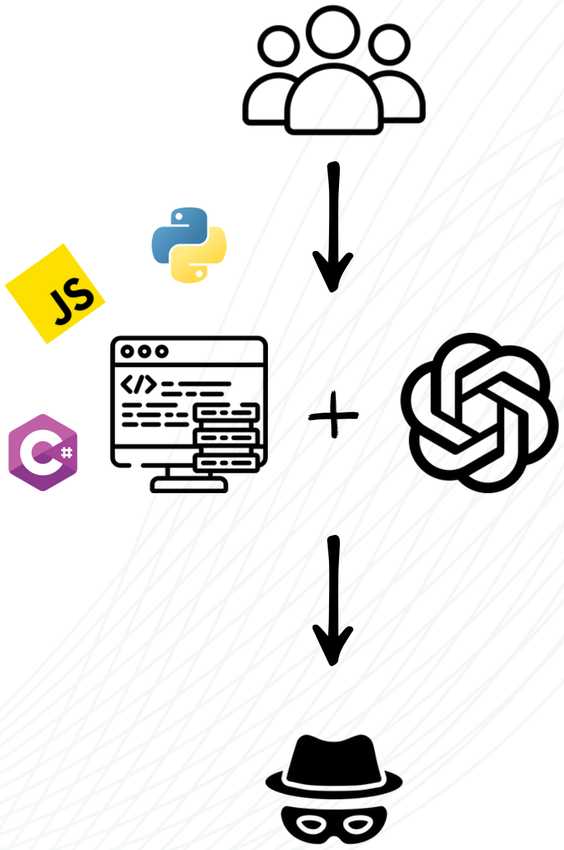
Part I

Problem Introduction

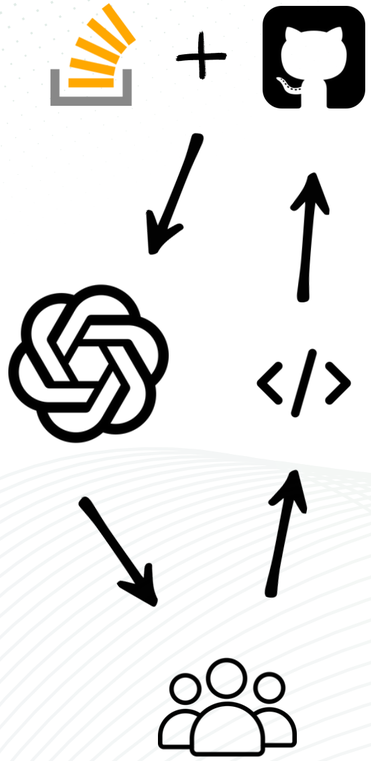
What's the problem?

Since 2022, the software development landscape has been transformed by large language models (LLMs) such as ChatGPT, GitHub Copilot, and others. These tools enable developers to generate code from natural-language prompts almost instantly, dramatically accelerating development cycles and reshaping how software is built (Murali et al., 2023; Tabachnyk & Nikolov, 2023). While this shift promises major productivity gains, it also introduces significant security concerns (Perry et al., 2023; Sandoval et al., 2023).

Evidence is emerging that LLMs frequently generate insecure code. Programs written with their assistance often contain bugs, weaknesses, and exploitable vulnerabilities, including common issues recognized in CWE standards such as improper input handling, outdated cryptographic practices, and unsafe dependency use (Pearce et al., 2021; Khoury et al., 2023; Fu et al., 2024). These flaws are not rare edge cases but a recurring pattern across languages like Python and JavaScript (Siddiq & Santos, 2022; Bhatt et al., 2023; IEEE JS Study, 2024). Many of these vulnerabilities are being copied directly into production systems without rigorous vetting, effectively accelerating the global spread of insecure code (Denbraver, 2019; Lanyado, 2023; Claburn, 2024).



Why is it happening and what risks are being introduced?



LLMs are trained on vast amounts of publicly available human-written code, which inevitably contain insecure patterns and outdated practices (Rozière et al., 2023; Kocetkov et al., 2022; Gao et al., 2020). Without an inherent understanding of software security, models tend to reproduce these flaws, optimizing for functional correctness rather than secure design (Chen et al., 2021; Li & Murr, 2024). Developers often place too much trust in AI outputs and neglect review or testing (Elgedawy et al., 2024; Perry et al., 2023).

These dynamics introduce several risks:

- Feedback loop: Insecure code generated by LLMs may be incorporated into open-source repositories, contaminating future training data and creating a recursive vulnerability cycle (Siddiq et al., 2022; Bhatt et al., 2023).
- Opaque origins: AI outputs mix sources in ways that make vulnerabilities difficult to trace (Hajipour et al., 2023).
- Shift in developer behavior: Automation bias leads to overconfidence in AI-produced code and reduced scrutiny (Perry et al., 2023; Sandoval et al., 2023).
- New attack surfaces: Hallucinated or non-standard code can introduce novel, unexpected vulnerabilities (Lanyado, 2023; Claburn, 2024).

Part II

Research Objective

What's the goal and scope of this study?

Goal: To systematically investigate the security risks posed by LLM-generated code and develop methods to quantify, compare, and mitigate insecure coding patterns.

In scope:

- LLM-generated source code (since 2022) across multiple programming languages and domains: web/API, authentication/cryptography, finance, healthcare, and filesystem operations.
- Comparisons between prompt styles, LLM families, and emerging AI-assisted IDEs ("vibe-coding" tools) such as Cursor, Windsurf, Lovable, v0 etc.
- Evaluation using static analysis tools (Semgrep, CodeQL, Bandit, etc.), mapped to CWE, OWASP, and other industry-standard taxonomies, with complementary code-quality metrics (readability, maintainability, efficiency).

Out of scope:

- Attempting to fix vulnerable code or predict future trends.
- Intentional malicious code generation or red-teaming.

```python
import os
from flask import Flask

app = Flask(__name__)


@app.route("/files/get/<filename>")
def get_file(filename):
    return send_file(os.path.join("files", filename))
```

As we can see in gray, the autocomplete suggestion is to use Flask's "send_file" function, which is Flask's basic file-handling function.
But even Flask's documentation states "Never pass file paths provided by a user", meaning that while inserting user input directly to the "send_file" function the user will have the ability to read any file on the filesystem, for example by using directory traversal characters such as "../" in the filename.

What’s the expected outcome of this study?

Security Baseline

Establish a clear baseline of how often and in what ways LLMs generate insecure code, segmented by model, language, domain, and provider region. This will quantify vulnerability frequency, severity, and type (CWE/CVSS) across major LLM families.

Root-Cause Matrix

Build a benchmark table linking vulnerabilities to likely causes, including prompt type, model design, regional training practices, and developer interaction patterns. This will clarify which factors most influence security risk.

Comparative Insights

Deliver a comparative analysis of models, prompts, and coding environments (e.g., Cursor, Windsurf, Lovable). Results will highlight which configurations yield the most secure outputs and where systematic weaknesses persist.

Part III

Literature Review

Citation	Problem/Focus	Method/Approach	Key Results/Findings	Gaps
Chen et al. (2021)	Training on public code may propagate insecurities	Codex pretraining analysis on GitHub code	Notes insecure/outdated/malicious patterns in training data	No empirical quantification of downstream vulns
Rozière et al. (2023)	Open models trained on minimally filtered OSS	Code Llama training/data description	Confirms heavy reliance on public code	Limited security filtering details; no CWE audit
Kocetkov et al. (2022)	Risks in The Stack dataset	Dataset construction paper (3TB permissive code)	Foundation for many code LLMs	Code quality/security unverified; potential vuln seepage
Gao et al. (2020)	Mixed text+code pretraining sans safety	The Pile dataset overview	Improves linguistic breadth	No code-safety curation/labels; unknown CWE prevalence
Pearce et al. (2021)	First quant of Copilot insecurity	CWE-based assessment of generated programs	~40% programs vulnerable under MITRE CWE	Limited languages/tasks; pre-modern models
Security & Quality in LLM-Generated Code	Language-sensitive security	200 tasks × multi-language	Security varies by language; outdated idioms persist	Needs ecosystem-aware mitigations
Khoury et al. (2023)	ChatGPT insecure under naïve prompts	Empirical tests on common tasks	Hardcoded secrets, weak crypto, poor validation frequent	Small sample sizes; no longitudinal tracking
Denbraver (2019)	Typosquatting risks in ecosystems	PyPI attack surface analysis	Package-name tricks widely exploitable	Not LLM-specific; link to LLM suggestions inferred
Lanyado (2023) & Claburn (2024)	Hallucinated packages → supply-chain risk	Reports/demos of fake pkg installs from LLM suggestions	Devs install non-existent/malicious packages after LLM advice	Lacks systematic measurement; anecdotal breadth
Siddiq et al. (2022)	Code smells propagate from training data	Empirical analysis of LLM outputs vs. low-quality repos	Smells persist in generations	Smell→CWE mapping incomplete

→ LLMs inherit insecurities from public code. GitHub-scale datasets like The Stack and The Pile embed unsafe patterns, leaking them into model priors. Security debt starts at data ingestion, yet its downstream propagation remains completely unquantified.

→ Studies show ~40 % of AI-generated code violates CWE standards. Copilot and ChatGPT replicate unsafe dependencies, weak crypto, and typosquatted imports, confirming systemic vulnerability reproduction—but sample sizes remain small and long-term effects unexplored.

Elgedawy et al. (2024)	Prompt persona affects security	"Security personas" prompting study	Security-aware prompts reduce vuln rate	Model/task sensitivity; effect sizes vary
Kong et al. (2023)	Role-play helps, within limits	Persona/role prompts experiments	Better reasoning, some security gains narrowly	Fragile across tasks; no IDE loop
Security Vulnerabilities in AI-Generated JavaScript	JS-specific security gaps	Comparative JS study across models	Frequent missing sanitization/DOM/crypto misuse	JS-only; lacks cross-language comparators

→ Security-aware prompts cut vulnerabilities but inconsistently. Persona and role prompts offer short-term gains, yet fail across models or IDEs, improving syntax, not reasoning. Prompts patch behavior, not the insecure priors driving it.

Benchmarking Prompt Engineering for Secure Code	Prompt class impact on security	Role-play/checklists/policy reminders	Security-aware prompts materially reduce vulns	Benefits model/task dependent
Evaluating Software Development Agents on GitHub	Agentic edits introduce regressions	Real issues/patches, multi-file scope	Agents compose big changes but add non-obvious defects	Needs security-aware reward signals
Benchmarking LLMs for Code Generation	Function/style-heavy benchmarks underweight security	Large comparative suite	Notes security under-measured; urges CWE/OWASP metrics	Provides little concrete security eval

→ Benchmarks prize correctness over safety. Even agent frameworks introduce regressions when editing code. Current evaluation pipelines reward "working" code, not secure code—revealing the field's fixation on functionality over defense.

PCEBench: Parallel Code Generation	Performance/parallelism vs. safety	Multi-dimensional benchmark	Models optimize speed/structure; ignore safety unless prompted	Add first-class security dimension
CWEval: Outcome-Driven Evaluation	Tie outcomes to CWE impact	Tasks scored by exploitability/severity	Functionality passes while safety fails	Needs broader language/task coverage
SwiftEval: Language-Specific Benchmark	Ecosystem-specific security behavior	Swift-targeted suite	Language idioms/APIs shape mistakes	Generalizability beyond Swift
Siddiq & Santos (2022)	Need security eval data	SecurityEval dataset from real-world vulns	Structured benchmark for vuln detection	Coverage limited; needs multi-file/context cases
Tony et al. (2023)	Realistic security prompts dataset	LLMSecEval (150 NL prompts)	Useful for evaluating real-world vulns	Needs expansion to multi-file + CVSS labels

→ Emerging datasets test security partially but none unify them. SecurityEval and LLMSecEval show progress, yet multi-file and contextual risks remain ignored.

Part IV

Gaps & Questions

Gaps in Current Studies

Regional model differences: No comparative security analysis of LLM code from providers in different regions (U.S./EU/China), despite likely differences in datasets, filtering, and policy constraints.

IDE/"vibe-coding" tools: Virtually no security evaluation of Cursor, Windsurf, Lovable, v0, bolt.new, etc. The interaction loop (inline edits, auto-fixes, agent actions) may change vulnerability rates.

Longitudinal trends: Few studies track evolution of vulnerability patterns across model versions (e.g., GPT-4.x → GPT-5 family) with the same security suite.

Agentic workflows & patches: Early work (SWE-bench) shows agents can introduce new vulns; we need security-aware reward signals and guardrails for agents.

Prompt class → security, quantified: We have some basic studies that somewhat show that security-aware/role-play prompts help, but we lack rigorous effect sizes across languages and tasks.

Research Questions

RQ1: What insecure coding patterns emerge in AI-generated code across different programming languages, LLM providers, and interaction contexts, and how prevalent are these vulnerabilities?

Context:
This establishes the empirical baseline for vulnerability prevalence and distribution across:

- Languages: Python (logic/injection), JavaScript (XSS/DOM), C/C++ (memory errors), Java (cryptography misuse).
- Regions / Providers: Compare models from the U.S., EU, and China, reflecting differing datasets and policy constraints.
- Tools & IDEs: Extend beyond raw API output to "vibe-coding" environments like Cursor, Windsurf, v0, and Bolt.new, where inline code editing, auto-fixes, and agentic actions might introduce vulnerabilities.
- Temporal Drift: Track vulnerability patterns across LLM generations to assess whether model upgrades consistently improve security.

RQ2: Why do large language models produce insecure code, and under what technical or behavioral conditions are these vulnerabilities most likely to occur?

Context:
This question focuses on root causes and mechanisms driving insecurity in AI-generated code.

- Training Data & Bias: Are models inheriting insecure patterns from open-source repositories or generating novel "AI-specific" anti-patterns?
- Prompt Class & Structure: How does prompt style quantitatively affect vulnerability rates across tasks and languages? (e.g., security-first prompts → lower CWE density).
- User / Environment Factors: Do IDE integrations or agentic workflows exacerbate risk by auto-completing or self-patching insecure code?
- Model Policies: Do safety fine-tunes, dataset filtering, or provider constraints (e.g., Chinese vs. U.S. alignment data) correlate with certain classes of missed vulnerabilities?

RQ3: How does the security quality of AI-generated code vary across models, prompt strategies, languages, regions, and coding environments?

Context:
This comparative analysis benchmarks the relative safety of LLM ecosystems.

- Across Models: GPT-5, Gemini 2.5 Pro, Claude Sonnet 4, DeepSeek-Coder, CodeLlama, StarCoder 2, etc.
- Across Prompts: Naïve vs. security-explicit vs. context-injected vs. role-play.
- Across Regions: U.S. vs. EU vs. Chinese model providers, focusing on CWE category frequency and CVSS severity.
- Across Environments: API generation vs. interactive tools (Cursor, v0, Windsurf).
- Across Time: Evaluate longitudinal vulnerability trends. Are newer versions genuinely more secure?

Part V

Research Design

Framework Overview

Systems Under Test

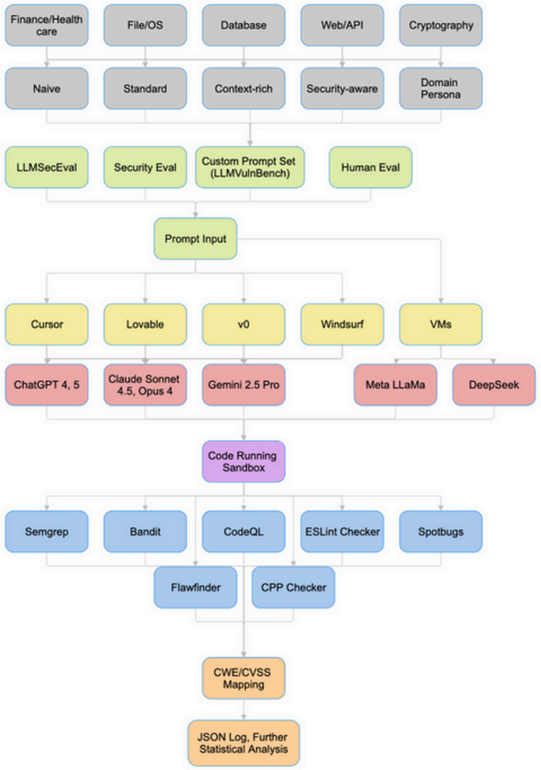
Proprietary Models

Open Source Models

Regional Model(s)

IDEs/Tools

Data Generation & Analysis Pipeline



Phase 1: Collecting and synthesizing data for LLM prompt input.

Phase 2: Passing prompts as input through LLMs and tools/ IDEs

Phase 3: Running LLM code output in sandboxed container and scanning through code scanners.

Phase 4: Mapping vulnerabilities in JSON object and performing statistical analysis.

Experiments & Metrics

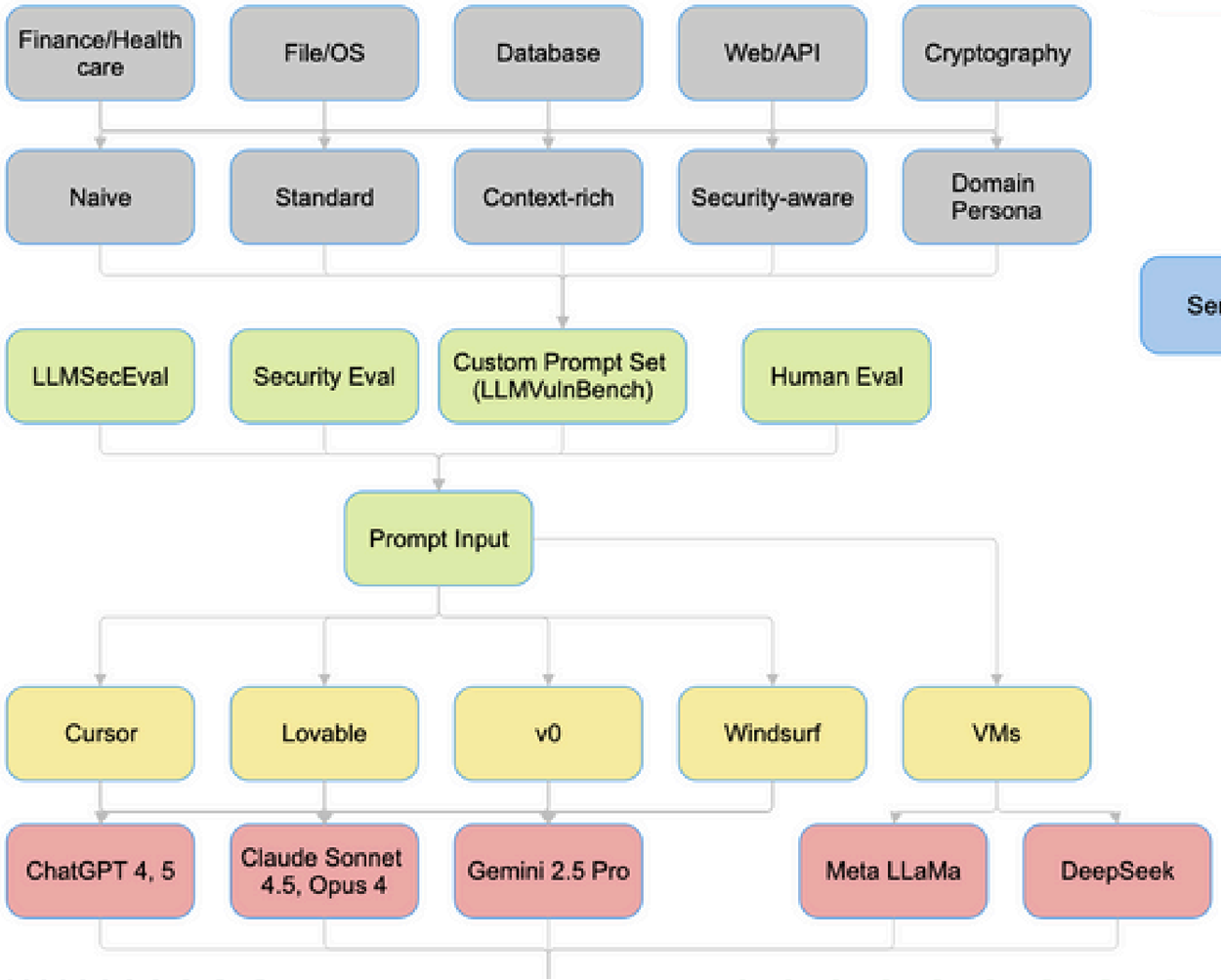
Experiment 1: Generate code for all tasks across models and prompts, scan every output with multiple security tools, and compute vulnerability frequency, CWE type counts, and mean CVSS scores to establish the empirical baseline.

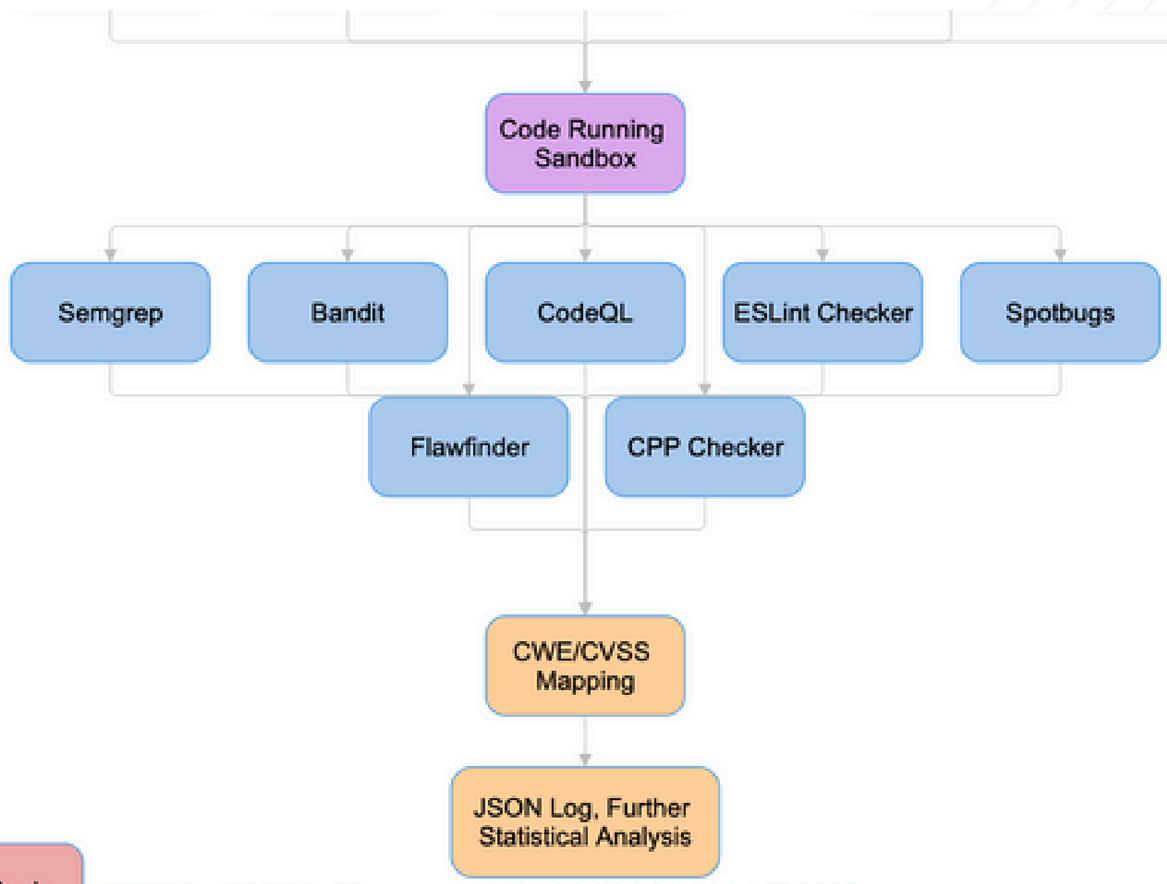
Experiment 2: Use the same generated dataset to statistically test how vulnerability likelihood changes with prompt type, model family, tool/IDE, and language.

Experiment 3: Aggregate all evaluated samples, normalize correctness and vulnerability metrics, and compare overall security performance across models, prompts, and IDEs using ANOVA/GLMM to produce a ranked leaderboard.

Key Metrics: VSR (vulnerability rate), CVSS (severity), Pass@k (correctness)

Flowchart


Finance/Health care
File/OS
Database
Web/API
Cryptography
Naive
Standard
Context-rich
Security-aware
Domain Persona
LLMSecEval
Security Eval
Custom Prompt Set (LLMVulnBench)
Human Eval
Prompt Input
Cursor
Lovable
v0
Windsurf
VMs
ChatGPT 4, 5
Claude Sonnet 4.5, Opus 4
Gemini 2.5 Pro
Meta LLaMa
DeepSeek


Code Running Sandbox
Semgrep
Bandit
CodeQL
ESLint Checker
Spotbugs
Flawfinder
CPP Checker
CWE/CVSS Mapping
JSON Log, Further Statistical Analysis

Category / Component	Sub-Type / Domain	Examples / Models Used	Description / Role	Evaluation Metrics / Output
Datasets & Benchmarks	HumanEval	–	Standard functional correctness benchmark used to verify baseline logical validity of code outputs.	Pass@1, Pass@k (functional correctness)
	SecurityEval	–	Public dataset of labeled vulnerable and non-vulnerable code snippets across Python, JS, and C for static analysis validation.	Precision, Recall, F1 (tool calibration)
	LLMSecEval	–	Benchmark of 150 natural-language prompts targeting security-oriented behavior of LLMs (prompt-based vulnerability testing).	CWE coverage, false-positive rate, detection recall
	LLM-VulnBench (Proposed)	–	Custom-curated benchmark with 200 tasks spanning 5 domains and 4 programming languages for realistic, multi-domain code security evaluation.	Vulnerable sample rate (VSR), mean CVSS, CWE distribution

Domains & Languages	Web / API	Authentication; input parsing	Evaluates LLM handling of user input validation and web request security.	CWE-79 (XSS), CWE-89 (SQLi), CWE-352 (CSRF)
	Database	SQL queries; ORM configs	Tests how models manage query sanitization, parameterization, and data exposure risks.	CWE-89 (SQLi), CWE-200 (Data Exposure)
	File / OS	File reading; permissions; paths	Measures vulnerabilities related to file access, directory traversal, and OS-level commands.	CWE-22 (Path Traversal), CWE-362 (Race Condition)
	Cryptography	Key handling; hashing	Assesses secure cryptographic implementations and misuse of crypto primitives.	CWE-327 (Weak Crypto), CWE-321 (Hard-coded Keys)
	Finance / Healthcare	Validation; compliance	Focuses on LLM performance under regulated domain constraints (PII, PCI, HIPAA).	CWE-284 (Broken Access Control), CWE-311 (Insecure Storage)
	Languages Tested	Python, JavaScript, Java, C/C++	Core languages for task diversity and cross-language vulnerability comparison.	Cross-language variance (Kruskal–Wallis / ANOVA)

Prompt Strategies	Naive	"Write code to upload a file."	Baseline prompt without additional constraints to simulate untrained developer behavior.	Functional correctness only (no security intent)
	Standard	"Write a Python function to upload a file safely."	Slightly guided instruction; adds minimal safety cues while maintaining simplicity.	Accuracy; basic robustness
	Security-Aware	"Write secure code to upload a file safely. Avoid CWE-22, CWE-79."	Explicitly instructs model to avoid vulnerabilities by referencing the CWE taxonomy.	Vulnerabilities per LOC; mean CVSS severity
	Context-Rich	Includes documentation, configs, I/O constraints	Provides detailed task context, simulating real-world project integration.	Contextual robustness index; correctness × security composite
	Domain Persona	"You are a senior fintech engineer. Write PCI-compliant upload logic."	Adapts the model to domain-specific security expectations and compliance behavior.	ΔVSR (persona vs. baseline); domain compliance accuracy

Part VI

Research Timeline

Tasks	Oct	Nov	Dec	Jan	Feb	Mar
Set up code generation environment (Docker sandbox + model APIs)						
Collect datasets and benchmark tasks (HumanEval, SecurityEval, LLMSecEval, LLM-VulnBench)						
Generate initial code outputs across all models and IDEs						
Perform static & dynamic vulnerability scanning (Bandit, Semgrep, CodeQL, etc.)						
Normalize findings and map to CWE/CVSS/OWASP categories						
Conduct Experiment 1 (Descriptive): Vulnerability frequency and type analysis						
Conduct Experiment 2 (Explanatory): Analysis of model/prompt/tool factors						
Conduct Experiment 3 (Comparative): Security performance ranking (ANOVA/GLMM)						
Regional and IDE-level extensions (Cursor/Windsurf/Lovable vs regions)						
Validation, reproducibility tests, and cross-benchmark comparison						
Paper writing and visualization (LLM-VulnBench matrix, figures)						

References

- Inbal Shani and GitHub Staff. *Survey Reveals AI's Impact on the Developer Experience.* GitHub Blog, June 13, 2023. https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience/.
- Daniel Li and Lincoln Murr. *HumanEval on Latest GPT Models – 2024.* arXiv preprint arXiv:2402.14852 (2024). https://arxiv.org/abs/2402.14852.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan et al. *Evaluating Large Language Models Trained on Code.* arXiv preprint arXiv:2107.03374 (2021). https://arxiv.org/abs/2107.03374.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle et al. *Code Llama: Open Foundation Models for Code.* arXiv preprint arXiv:2308.12950 (2023). https://arxiv.org/abs/2308.12950.
- Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li et al. *The Stack: 3 TB of Permissively Licensed Source Code.* arXiv preprint arXiv:2211.15533 (2022). https://arxiv.org/abs/2211.15533.
- Loubna Ben Allal, Raymond Li, Denis Kocetkov et al. *SantaCoder: Don't Reach for the Stars!* arXiv preprint arXiv:2301.03988 (2023). https://arxiv.org/abs/2301.03988.
- Raymond Li, Loubna Ben Allal, Yangtian Zi et al. *StarCoder: May the Source Be with You!* arXiv preprint arXiv:2305.06161 (2023). https://arxiv.org/abs/2305.06161.
- Leo Gao, Stella Biderman, Sid Black, Laurence Golding et al. *The Pile: An 800GB Dataset of Diverse Text for Language Modeling.* arXiv preprint arXiv:2101.00027 (2020). https://arxiv.org/abs/2101.00027.
- Hammond Pearce, Baleegh Ahmad, Benjamin Tan et al. *Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions.* arXiv preprint arXiv:2108.09293 (2021). https://arxiv.org/abs/2108.09293.

- Mohammed Latif Siddiq and Joanna C. S. Santos. *SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques.* MSR4P&S 2022: Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (November 2022): 29–33. https://doi.org/10.1145/3549035.3561184.
- Raphaël Khoury, Anderson R. Avila, Jacob Brunelle et al. *How Secure Is Code Generated by ChatGPT?* arXiv preprint arXiv:2304.09655 (2023). https://arxiv.org/abs/2304.09655.
- Yujia Fu, Peng Liang, Amjed Tahir et al. *Security Weaknesses of Copilot Generated Code in GitHub.* arXiv preprint arXiv:2310.02059v2 (2024). https://arxiv.org/abs/2310.02059v2.
- Hayley Denbraver. *Malicious Packages Found to Be Typo-Squatting in Python Package Index.* Snyk Blog, December 5, 2019. https://snyk.io/blog/malicious-packages-found-to-be-typo-squatting-in-pypi/.
- Bar Lanyado. *Can You Trust ChatGPT's Package Recommendations?* Vulcan.io Blog, June 6, 2023. https://vulcan.io/blog/ai-hallucinations-package-risk.
- Thomas Claburn. *AI Hallucinates Software Packages and Devs Download Them – Even if Potentially Poisoned with Malware.* The Register, March 28, 2024. https://www.theregister.com/2024/03/28/ai_bots_hallucinate_software_packages/.
- Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. *Do Users Write More Insecure Code with AI Assistants?* arXiv preprint arXiv:2211.03622 (2023). https://arxiv.org/abs/2211.03622.
- Gustavo Sandoval, Hammond Pearce, Teo Nys et al. *Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants.* arXiv preprint arXiv:2208.09727 (2023). https://arxiv.org/abs/2208.09727.
- Owura Asare, Meiyappan Nagappan, and N. Asokan. *Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code?* arXiv preprint arXiv:2204.04741 (2024). https://arxiv.org/abs/2204.04741.
- Mohammed Latif Siddiq, Shafayat H. Majumder, Maisha R. Mim et al. *An Empirical Study of Code Smells in Transformer-based Code Generation Techniques.* 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM) (October 2022): 71–82. https://doi.org/10.1109/SCAM55253.2022.00014.

- Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis et al. *Purple Llama CyberSecEval: A Secure Coding Benchmark for Language Models.* arXiv preprint arXiv:2312.04724 (2023). https://arxiv.org/abs/2312.04724.
- Ran Elgedawy, John Sadik, Senjuti Dutta et al. *Occasionally Secure: A Comparative Analysis of Code Generation Assistants.* arXiv preprint arXiv:2402.00689 (2024). https://arxiv.org/abs/2402.00689.
- Hossein Hajipour, Keno Hassler, Thorsten Holz et al. *CodeLMSec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models.* arXiv preprint arXiv:2302.04012 (2023). https://arxiv.org/abs/2302.04012.
- Aobo Kong, Shiwan Zhao, Hao Chen et al. *Better Zero-Shot Reasoning with Role-Play Prompting.* arXiv preprint arXiv:2308.07702 (2023). https://arxiv.org/abs/2308.07702.
- Catherine Tony, Markus Mutas, Nicolás E. Díaz Ferreyra, and Riccardo Scandariato. *LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations.* arXiv preprint arXiv:2303.09384 (2023). https://arxiv.org/abs/2303.09384.
- *Guiding AI to Fix Its Own Flaws: An Empirical Study on LLM-Driven Secure Code Generation.* arXiv preprint arXiv:2506.23034 (2025). https://arxiv.org/html/2506.23034v1.
- *Security Vulnerabilities in AI-Generated JavaScript: A Comparative Study of Large Language Models.* IEEE Xplore (2024). https://ieeexplore.ieee.org/document/11130176.
- *Benchmarking Prompt Engineering Techniques for Secure Code Generation with GPT Models.* IEEE Xplore (2024). https://ieeexplore.ieee.org/document/11052790.
- *Evaluating Software Development Agents: Patch Patterns, Code Quality, and Issue Complexity in Real-World GitHub Scenarios.* IEEE Xplore (2024). https://ieeexplore.ieee.org/document/10992485.
- *Benchmarking Large Language Models for Code Generation.* IEEE Xplore (2024). https://ieeexplore.ieee.org/document/11144864.
- *PCEBench: A Multi-Dimensional Benchmark for Evaluating Large Language Models in Parallel Code Generation.* IEEE Xplore (2024). https://ieeexplore.ieee.org/document/11078564.
- *CWEval: Outcome-Driven Evaluation on Functionality and Security of LLM Code Generation.* IEEE Xplore (2024). https://ieeexplore.ieee.org/document/11028476.