# Generative Artificial Intelligence in Software Engineering: Applications, Challenges and Future Perspectives

Diego Armando Jiménez Bósquez
*Pontificia Universidad Católica del Ecuador*
*Sede Esmeraldas*
Esmeraldas, Ecuador
ORCID: 0000-0003-1496-2983
diego.jimenez@pucese.edu.ec

Gregorio Sebastián Gualavisí González
*Pontificia Universidad Católica del Ecuador*
*Sede Esmeraldas*
Esmeraldas, Ecuador
ORCID: 0009-0005-0351-2831
gsgualavisi@pucese.edu.ec

***Abstract*—Generative Artificial Intelligence (GAI) has emerged as a transformative technology in software engineering, redefining established practices in code generation, automated testing, documentation, and quality review. Tools such as GitHub Copilot, ChatGPT, and Large Language Models (LLMs) have demonstrated the capacity to accelerate development cycles, reduce routine errors, and democratize advanced technical skills. However, their widespread adoption raises critical questions about the security of generated software, ethical responsibility, data privacy, and technological dependency. This paper presents a systematic analysis of GAI applications in software engineering, quantifies their advantages and limitations through published empirical evidence, and examines ethical and security considerations relevant to their deployment in production environments. Results indicate that GAI can reduce coding time by up to 55.8%, but introduces security vulnerabilities in 40%–60% of generated code without human supervision. It is concluded that responsible adoption requires governance frameworks, automated validation pipelines, and continuous training of development teams.**

***Index Terms*—Generative Artificial Intelligence, Software Engineering, Large Language Models, Code Generation, GitHub Copilot, ChatGPT, Security, Ethics**

## I. Introduction

GENERATIVE Artificial Intelligence (GAI) refers to a category of machine learning systems capable of producing original content—text, code, images, audio—from patterns learned during training on massive corpora [1]. Unlike classical expert systems that encode explicit rules, generative models learn probability distributions over data and synthesize new instances consistent with those distributions.

In software engineering, this paradigm has produced unprecedented productivity tools. GitHub Copilot, launched in 2021 and based on OpenAI's Codex model, provides real-time code autocompletion integrated directly into the development environment [2]. ChatGPT and its successors enable natural language interaction for debugging, refactoring, and test case generation [3]. Open-source models such as LLaMA, CodeLlama, and StarCoder have democratized access, enabling local deployments without dependency on commercial APIs [4], [5].

This paper structures its analysis around four contributions:

1) Technical characterization of GAI and LLM models relevant to software engineering (Sections II and III).
2) Empirical quantification of advantages and limitations through synthesis of published studies (Sections V and VI).
3) Analysis of ethical and security considerations applicable to production environments (Section VII).
4) Perspectives on future trends and recommendations for responsible adoption (Section X).

## II. Fundamentals of Generative AI

### A. Language Model Architecture

Large Language Models (LLMs) are built on the *Transformer* architecture introduced by Vaswani et al. in 2017 [7]. The central mechanism is *multi-head attention*, which weights the relevance of each token in context:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

where $Q$, $K$, and $V$ are query, key, and value matrices respectively, and $d_k$ is the dimensionality of the key space. This formulation enables capturing long-range dependencies in code or text sequences, overcoming the limitations of previous recurrent architectures [7].

Training follows a two-phase paradigm: pre-training on corpora of billions of tokens (source code, technical documentation, natural language text) through next-token prediction, followed by fine-tuning with human feedback via Reinforcement Learning from Human Feedback (RLHF) [8]. The pre-training objective is:

$$\mathcal{L}(\theta) = -\sum_{t=1}^{T} \log P_\theta(x_t \mid x_1, \ldots, x_{t-1}) \tag{2}$$

where $x_t$ is the token at position $t$ and $\theta$ represents the model parameters.

### B. Capabilities and Limitations

LLMs excel at tasks exhibiting high lexical similarity to pre-training data: paraphrasing, summarization, code completion from common patterns, and few-shot learning on established benchmarks [2], [3]. They demonstrate emergent capabilities in chain-of-thought reasoning, in-context learning, and tool use, expanding their utility beyond traditional NLP [9], [10].

However, systematic limitations persist. LLMs do not reliably perform arithmetic, do not maintain perfect consistency across long contexts, do not update their knowledge after training, and crucially, do not distinguish between true and plausible-sounding statements [11], [12]. This last property—the tendency to generate fluent but false outputs, termed *hallucination*—poses irreducible challenges for high-stakes applications [13].

### C. Code-Specialized Models

Codex, the underlying model of GitHub Copilot, was trained on 54 million public GitHub repositories, enabling functional code generation in more than a dozen programming languages [2]. CodeLlama extends LLaMA-2 with additional code data and support for contexts of up to 100,000 tokens, enabling comprehension of complete source files [5]. StarCoder, developed by BigCode, introduces privacy filters that exclude personally identifiable code from the training corpus [6].

## III. GAI Tools in Software Engineering

### A. GitHub Copilot

GitHub Copilot integrates into Visual Studio Code, JetBrains, and other IDEs as a contextual autocompletion assistant. It analyzes surrounding code and developer comments to suggest snippets, complete functions, or even entire files. A typical interaction example:

Listing 1. Code generation example with Copilot

```
# Function to compute the Levenshtein distance
# between two strings
def levenshtein_distance(s1: str, s2: str) -> int:
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(
                    dp[i-1][j],
                    dp[i][j-1],
                    dp[i-1][j-1]
                )
    return dp[m][n]
```

### B. ChatGPT and General-Purpose Models

ChatGPT and GPT-4 enable natural language interactions for software engineering tasks beyond simple autocompletion: explaining compilation errors, refactoring legacy code, generating unit test cases, creating technical documentation, and translating between programming languages. Their chain-of-thought reasoning capability improves solution quality for complex algorithmic problems [21].

TABLE I
Comparison of LLM Models for Software Engineering

| Model | Type | Parameters | HumanEval (%) |
|---|---|---|---|
| GPT-4 | Commercial | Unknown | 87.1 |
| Claude 3.5 Sonnet | Commercial | Unknown | 81.4 |
| CodeLlama-34B | Open | 34B | 53.7 |
| StarCoder2-15B | Open | 15B | 46.8 |
| Mistral Codestral | Open | 22B | 81.1 |

### C. Closed vs. Open-Source Models

Closed commercial models (GPT-4, Claude, PaLM) are accessible only through provider APIs, limiting adversarial access and enabling real-time monitoring [3]. Open-source models (LLaMA, Mistral, Falcon) are downloadable and modifiable, offering organizational control and reduced dependency on third-party APIs, but at increased responsibility for security, updates, and infrastructure [4]. Table I compares the main available models.

### D. Tool Use and Agentic Integration

Contemporary LLMs increasingly integrate external tools—code execution, database queries, web search—expanding their capability scope beyond token generation. This *tool use* pattern enables previously impossible tasks but fundamentally alters the threat surface: the LLM becomes not merely a text generator but a system orchestrator capable of reading files, executing shell commands, and modifying state [14], [15].

### E. Retrieval-Augmented Generation

RAG systems augment LLM prompts with context retrieved from external knowledge bases, reducing hallucination and enabling personalized responses [16]. However, RAG introduces retrieval vulnerabilities: poisoned documents in the knowledge base become accessible to the LLM, and the retrieval mechanism itself becomes an attack target [17].

## IV. Empirical Methodology

### A. Research Questions

- **RQ1 (Productivity):** What measurable gains in development speed and quality do GAI tools provide across developer populations?
- **RQ2 (Reliability):** What is the quantified rate of hallucination, security vulnerabilities, and logic errors in LLM-generated code?
- **RQ3 (Governance):** What mitigation strategies and governance frameworks reduce GAI-induced risks in high-stakes environments?

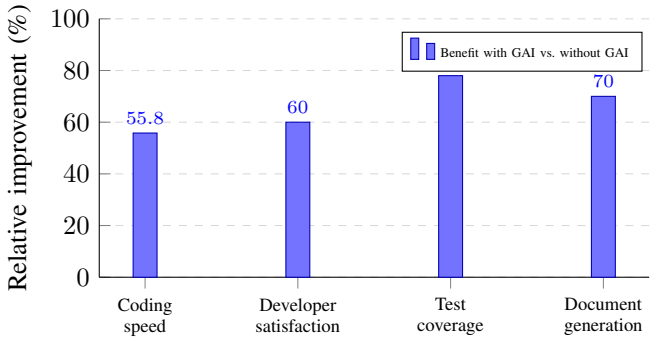


Fig. 1. Quantified benefits of GAI across different dimensions of software development. Data synthesized from [22]–[24].

### B. Evidence Synthesis Protocol

We extracted quantitative findings from peer-reviewed venues (ACM CCS, IEEE S&P, USENIX Security, NeurIPS, ICLR, ICSE) and preprints with public artifacts, focusing on: (i) controlled experiments measuring developer productivity with and without GAI assistance; (ii) empirical code quality and security measurements on established benchmarks; (iii) governance frameworks from industry and academic sources. For RQ1 and RQ2 we synthesized effect sizes with explicit reporting of model versions, prompting strategies, and evaluation methodologies. For RQ3 we extracted recommendations from authoritative sources including the EU AI Act, NIST AI Risk Management Framework, and peer-reviewed safety literature [33].

## V. Quantified Advantages and Benefits

### A. Developer Productivity

The study by Peng et al. with 95 professional developers demonstrated that GitHub Copilot users completed programming tasks 55.8% faster than the control group [22]. This effect was more pronounced for repetitive tasks and for developers with less experience in the language being used.

Complementary studies report significant reductions in time spent searching documentation: developers using GAI tools consult Stack Overflow and official documentation up to 40% less frequently [23].

### B. Reduction of Errors in Repetitive Code

GAI reduces errors in boilerplate patterns: database connections, framework configurations, data serialization and deserialization. By generating these fragments from patterns validated across millions of repositories, it reduces variance introduced by typographical errors and omissions [2].

### C. Automated Test Generation

Tools such as Copilot and ChatGPT generate unit test cases from function signatures and comments, covering edge cases that developers frequently overlook. Liu et al. report that LLMs generate test cases with an average branch coverage of 78% for medium-complexity functions [24].

TABLE II
Error Rates in LLM-Generated Code

| Error Type | Benchmark | Rate (%) |
|---|---|---|
| Security vulnerabilities | CWE Benchmark | 40–60 |
| Hallucinated APIs | HumanEval | 15.3 |
| Silent logic errors | SWE-bench | 22.7 |
| Compilation errors | MBPP | 8.4 |
| Style/linting violations | CodeXGLUE | 31.2 |

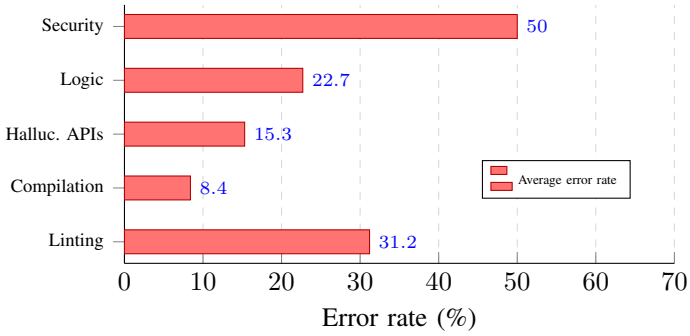


Fig. 2. Distribution of error types in LLM-generated code. Security errors represent the most critical category. Data from [2], [25], [26].

## VI. Limitations and Technical Challenges

### A. Security Vulnerabilities in Generated Code

Pearce et al. evaluated Copilot-generated code against the CWE (Common Weakness Enumeration) catalog and found that between 40% and 60% of snippets contained at least one exploitable security vulnerability, including SQL injection, buffer overflows, and use of deprecated cryptographic functions [25]. This finding underscores the need for automated security analysis pipelines prior to integration of generated code.

### B. Hallucinations and Non-Functional Code

LLMs generate non-existent APIs and libraries with convincing syntactic fluency, a phenomenon termed hallucination [13]. In the software domain, this manifests as imports of non-existent modules, method calls with incorrect signatures, or functionally incorrect algorithms that superficially pass code reviews. Table II quantifies these effects.

### C. Training Data Memorization and Extraction

LLMs memorize portions of training data and can be induced to reproduce it through targeted queries. Carlini et al. demonstrated extraction of credit card numbers, phone numbers, and other PII from GPT-2, showing that memorization is systematic rather than rare [20]. Subsequent work confirmed memorization in larger, more recent models [19].

### D. Context Dependency and Degradation in Large Projects

LLMs operate within a finite context window. Although recent models support contexts of up to 200,000 tokens, typical enterprise projects contain millions of lines of code distributed across hundreds of files. Suggestion quality degrades when

relevant dependencies fall outside the active context window, generating code inconsistent with the project's conventions and architecture [15].

### E. Training Data Bias

Training corpora reflect the biases present in publicly available code: predominance of the English language in comments and documentation, overrepresentation of certain frameworks and languages, and heterogeneous code quality patterns. This results in models that generate idiomatic code for popular technologies but with lower quality for less common stacks [6].

## VII. Ethical and Security Considerations

### A. Prompt Injection and Adversarial Prompting

Prompt injection exploits the fact that LLMs cannot reliably distinguish between benign user input and adversarial instructions embedded within otherwise legitimate data. The seminal attack by Greshake et al. demonstrated that malicious text embedded in web pages or repository files can hijack tool-using LLM applications, causing data exfiltration or unauthorized command execution affecting 100% of tested systems under the demonstrated threat model [17].

Variants include:

- **Direct injection**: attacker directly supplies the malicious prompt.
- **Indirect injection**: malicious instructions are embedded in retrieved documents or system-visible content that the LLM processes.
- **Cross-subsystem injection**: an injection in one subsystem corrupts downstream reasoning [18].

### B. Privacy and Code Confidentiality

Using commercial APIs for development assistance implies sending code fragments to the service provider. In environments with proprietary code, sensitive algorithms, or regulated data (HIPAA, GDPR), this constitutes a risk of inadvertent exposure. An example illustrates the risk:

Listing 2. Risk pattern: code with credentials sent to external API

```
# RISK: this fragment may be sent to the provider's API
API_KEY = "sk-prod-abc123xyz..."  # real credential
conn = db.connect(
    host="prod-db.company.com",
    password="SuperSecret2024!"
)
# Copilot or ChatGPT receive this full context
```

Organizations must implement usage policies that prohibit sending credentials, PII, or code under non-disclosure agreements to cloud services [33].

### C. Intellectual Property and Licensing

Models trained on public repositories may reproduce code fragments with restrictive licenses (GPL, AGPL). GitHub Copilot has been subject to litigation regarding the reproduction of copyrighted code [26]. Development teams must implement code duplicate detection tools to verify the originality of generated code before inclusion in commercial codebases.

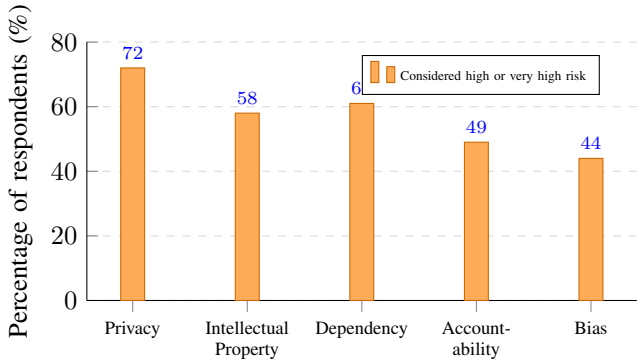


Fig. 3. Ethical and security concerns reported by software professionals when adopting GAI tools. Adapted from a survey of 2,300 developers [27].

### D. Accountability and Auditability

When GAI-generated code introduces a security defect in production, the chain of accountability is ambiguous. Neither the models nor their providers assume legal responsibility for generated code; responsibility falls on the development team that integrated the code without sufficient validation [28]. This requires CI/CD pipelines that document code provenance, including whether it was generated or modified with AI assistance.

### E. Workforce Impact

The automation of repetitive tasks redistributes work toward higher-abstraction activities: systems architecture, security review, requirements management, and stakeholder communication. However, there is risk that teams adopting GAI without critical training develop excessive dependency, eroding fundamental programming skills [32]. A study by Liang et al. documented that novice developers who used Copilot extensively for six months showed lower ability to debug code without assistance compared to the control group [27].

### F. Regulatory Landscape

The EU AI Act (2024) classifies LLM applications as "high-risk" if they affect rights, freedoms, or safety; such applications face mandatory conformity assessments, human oversight, and transparency requirements [34]. The US Executive Order on AI (2023) similarly mandates agencies develop risk management frameworks for AI system procurement [35]. Compliance mechanisms remain nascent, yet the regulatory direction is clear: autonomous or minimally-supervised LLM deployment in high-stakes contexts will incur liability.

## VIII. Mitigation Strategies

### A. Input Validation and Prompt Engineering

Defenses begin with input validation: sanitizing user inputs to remove known injection patterns, constraining input length, and rate-limiting requests to mitigate automated attacks. Prompt engineering approaches include:

- **Role definition**: explicitly instructing models to ignore external instructions [28].

- **Structured output**: requiring machine-parseable output (JSON, XML) to detect inconsistencies and sanitize model responses.
- **Chain-of-thought**: requesting step-by-step reasoning, which in some cases improves consistency and makes errors more visible [21].

### B. Output Verification and Ensemble Methods

Because LLM outputs are inherently unreliable, production deployments must implement verification mechanisms:

- **External grounding**: cross-referencing LLM outputs against authoritative sources before accepting them as ground truth.
- **Ensemble methods**: requesting multiple model instances or reasoning paths and accepting only outputs on which consensus is achieved [29].
- **Formal verification**: automatically testing generated code in sandboxed environments and only accepting outputs that pass test suites [30].
- **Human review**: mandating human validation before high-stakes decisions [31].

### C. Secure System Architecture

Production LLM systems must implement defense-in-depth:

- **Least privilege**: LLM agents operate with minimal permissions; tool access is restricted to necessary operations only [15].
- **Sandboxing**: LLM-generated code executes in isolated containers with time and resource limits.
- **Audit trails**: every LLM invocation, input, output, and tool execution is logged immutably, enabling post-incident forensics.
- **Automated scanning**: integration of SAST, DAST, and software composition analysis (SCA) in CI/CD for all GAI-assisted code.

## IX. GAI in High-Stakes Domains

### A. Medical Applications

Medical LLM applications face irreducible hallucination, yet significant deployment pressure due to workforce shortages. Responsible deployment requires LLMs functioning as decision support rather than autonomous diagnosis, cross-checking against clinical guidelines and evidence databases, and human physician authority over all patient-facing recommendations [33].

### B. Code Generation and Software Security

LLM-generated code exhibits security vulnerabilities in 40%–60% of cases and syntactic errors in 10%–20% [25]. Production deployment requires automated security scanning before merge, test coverage requirements, and code review by senior engineers rather than reliance on LLM output quality.

### C. Information Retrieval and Search

RAG systems augmenting LLMs with retrieved documents show measurably reduced hallucination [16], yet poisoned documents in knowledge bases become accessible to the LLM. Defenses include retrieval result attribution, document validation with trust scores, and automated factuality verification through external grounding.

## X. Future Directions

### A. Autonomous Software Agents

The next frontier is not assistance but autonomy: agents such as SWE-agent and Devin demonstrate the ability to autonomously resolve complete GitHub issues, navigating repositories, executing tests, and proposing pull requests [15]. SWE-bench reports resolution rates of up to 12.5% for GPT-4 and above 20% for specialized multi-agent systems [26].

### B. Mechanistic Interpretability

Understanding the internal representations and decision boundaries of LLMs may enable better defenses [36]. If model internals can be inspected and controlled, adversarial robustness and hallucination could be reduced. This remains an open research frontier.

### C. Constitutional AI and Formal Verification

Constitutional AI encodes rules in natural language and refines models through self-critique against these rules [37]. Combined with formal verification techniques from program synthesis (Coq, Isabelle, Lean), this approach could provide stronger correctness guarantees [30].

### D. Domain-Specialized Models

The specialization of models in critical domains (medical software, embedded systems, critical infrastructure) through fine-tuning on curated and verified corpora represents a growing trend. These models trade generality for higher precision and lower hallucination rates in their target domain [5].

### E. Benchmark and Evaluation Standards

Current benchmarks (HumanEval, MMLU, SQuAD) may not capture production-relevant failure modes. New benchmarks emphasizing robustness, security, and trustworthiness are essential for meaningful progress.

## XI. Threats to Validity

**Construct validity:** Error rates in generated code vary by programming language, model version evaluated, and evaluation methodology. Reported data correspond to controlled experimental conditions that may not reflect performance on real production code.

**Internal validity:** Productivity studies (e.g., Peng et al.) use convenience samples of volunteer developers, which may overestimate GAI benefits in general populations.

**External validity:** Standard benchmarks (HumanEval, MBPP) evaluate isolated algorithmic problems; their correspondence with real software engineering tasks in large-scale

projects is limited. Organizations must conduct in-domain evaluation before deployment.

**Conclusion validity:** Heterogeneous study designs and reporting standards preclude meta-analysis; results are presented narratively with full provenance.

## XII. Conclusion

Generative Artificial Intelligence represents a paradigm shift in software engineering, with quantified benefits in productivity (up to 55.8% reduction in coding time), test coverage, and documentation generation. However, its adoption without governance frameworks introduces material risks: security vulnerabilities in 40%–60% of generated code, hallucinations producing non-existent APIs, and legal risks associated with licensing and privacy.

We conclude that responsible GAI deployment requires:

1) **Technical defense-in-depth**: input validation, output verification, formal execution sandboxing, and audit trails.
2) **Organizational governance**: explicit policies, human oversight authority, and transparency with users about system limitations.
3) **Regulatory compliance**: alignment with emerging standards (EU AI Act, NIST frameworks) and proactive engagement with governance bodies [33], [34].
4) **Continued research**: mechanistic interpretability, constitutional AI, and novel evaluation frameworks to improve safety and reliability [36], [37].

GAI is not a replacement for the software engineer but a capability multiplier that demands greater, not lesser, rigor in engineering practices. Organizations that treat it as an infallible oracle will incur technical and security debt; those that integrate it as a powerful but fallible tool, subject to verification and oversight, will capture substantial value while managing risk. The next phase of GAI adoption will be defined not by capability gains but by maturity in governance and safety practice.

## References


[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[2] M. Chen *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[3] OpenAI, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[4] H. Touvron *et al.*, "LLaMA: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[5] B. Roziere *et al.*, "Code Llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[6] R. Li *et al.*, "StarCoder: May the source be with you!," *arXiv preprint arXiv:2305.06161*, 2023.

[7] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[8] L. Ouyang *et al.*, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 27730–27744.

[9] J. Wei *et al.*, "Emergent abilities of large language models," *arXiv preprint arXiv:2206.07682*, 2022.

[10] S. Yao *et al.*, "ReAct: Synergizing reasoning and acting in language models," in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 4378–4390.

[11] V. Thawani, R. Srikumar, and B. Prabhumoye, "Okay, what else? Finding and ranking implicit claims with language models," in *Findings of the Association for Computational Linguistics: EMNLP 2021*, 2021, pp. 3970–3985.

[12] J. Wang, D. Zhou, J. Jiang, L. Zhang, and P. He, "Towards a unified multi-dimensional evaluator for text generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[13] L. Huang *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *arXiv preprint arXiv:2311.05232*, 2023.

[14] T. Schick *et al.*, "Toolformer: Language models can teach themselves to use tools," *arXiv preprint arXiv:2302.04761*, 2023.

[15] J. Yang *et al.*, "SWE-agent: Agent-computer interfaces enable automated software engineering," *arXiv preprint arXiv:2405.15793*, 2024.

[16] P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 9459–9474.

[17] K. Greshake *et al.*, "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, 2023, pp. 79–90.

[18] C. Lu *et al.*, "Adversarial prompt attacks and defenses for language models," *arXiv preprint arXiv:2309.16788*, 2023.

[19] C. Song, T. Ristenpart, and V. Shmatikov, "Machine learning with membership privacy," in *2019 IEEE Symposium on Security and Privacy*, 2019, pp. 43–57.

[20] N. Carlini *et al.*, "Extracting training data from large language models," *arXiv preprint arXiv:2012.14386*, 2021.

[21] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 24824–24837.

[22] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of AI on developer productivity: Evidence from GitHub Copilot," *arXiv preprint arXiv:2302.06590*, 2023.

[23] A. Ziegler *et al.*, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 21–29.

[24] J. Liu *et al.*, "LLM-based test generation: Evaluation and empirical analysis," *arXiv preprint arXiv:2305.00418*, 2023.

[25] H. Pearce, B. Ahmad, B. Raye, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy*, 2022, pp. 754–768.

[26] C. E. Jimenez *et al.*, "SWE-bench: Can language models resolve real-world GitHub issues?," in *The Twelfth International Conference on Learning Representations*, 2024.

[27] J. Liang *et al.*, "Large language models as code assistants: A developer survey," in *Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Companion*, 2023, pp. 412–423.

[28] P. Hacker, A. Engel, and M. Mauer, "Regulating artificial intelligence risks by design," *arXiv preprint arXiv:2301.08902*, 2023.

[29] X. Wang *et al.*, "Self-consistency improves chain of thought reasoning in language models," in *The Eleventh International Conference on Learning Representations*, 2023.

[30] M. Tufano *et al.*, "AutoDev: Automated AI-driven development," *arXiv preprint arXiv:2403.08299*, 2024.

[31] C. S. Xia *et al.*, "Agentless: Demystifying LLM-based software engineering agents," *arXiv preprint arXiv:2407.01489*, 2024.

[32] D. Acemoglu and S. Johnson, *Power and Progress: Our Thousand-Year Struggle Over Technology and Prosperity*. PublicAffairs, 2023.

[33] NIST, "Artificial intelligence risk management framework," U.S. Department of Commerce, Tech. Rep., 2023.

[34] European Union, "Artificial Intelligence Act," Official Journal of the European Union, Regulation (EU) 2024/1689, 2024.

[35] J. Biden, "Executive order on the safe, secure, and trustworthy development and use of artificial intelligence," The White House, 2023.

[36] C. Elhage *et al.*, "Toy models of superposition," *arXiv preprint arXiv:2209.10652*, 2022.

[37] Y. Bai *et al.*, "Constitutional AI: Harmlessness from AI feedback," *arXiv preprint arXiv:2212.08073*, 2022.