Applications of Large Language Models in Computer Programming: A Research-Oriented Survey

Introduction

Large Language Models (LLMs) have rapidly evolved into essential tools within modern software engineering. Their ability to interpret context, generate code, detect bugs, refactor complex software structures, and provide semantic understanding has transformed both academic research and industrial software development. Before exploring concrete examples of LLM-based applications, this study introduces the fundamental role of LLMs in programming workflows, setting the stage for detailed examination of real-world use cases and systems. The following sections provide categorized examples of how LLMs are applied across multiple areas of programming, including code generation, documentation, debugging, static analysis, testing, and autonomous agent-driven software development.

1. Code Generation and Autocompletion

LLM-powered code generation systems synthesize executable code from natural language instructions or partial code fragments. Examples include GitHub [1] Copilot [2], Amazon CodeWhisperer [3] , and Tabnine[4] .

2. Automated Code Documentation

Tools like Mintlify Doc Writer [5] and Sourcery AI [6] generate human-readable documentation by analyzing program structure, improving code maintainability.

3. Debugging and Error Explanation

Systems such as Copilot Chat[7], Replit AI Debugger[8], and JetBrains AI Assistant[9] interpret error logs and produce actionable debugging steps.

4. Static Code Analysis

Semgrep[10] + AI, Snyk Code AI[11], and SonarLint AI enhance traditional rule-based static analysis with semantic reasoning and natural-language vulnerability explanations.

5. Test Generation

Diffblue Cover[12], Copilot Test Generation, and TestSigma AI automatically construct unit and integration tests, improving coverage and reliability.

6. Semantic Code Search

Sourcegraph Cody[13], Google AI Code Search, and IBM Watson Code Navigator[14] support large-scale repository search using embedding-based semantics.

7. Autonomous Bug-Fixing Agents

OpenAI SWE-Agent[15], Repairnator[16], and Meta CodeCompose autonomously detect defects and generate patches.

8. DSL Interpretation

Systems like Stripe's natural-language payment pipelines, NVIDIA Isaac GPT[17], and OpenAI's function-calling architecture translate domain-specific commands into executable code.

9. Refactoring Assistants

Sourcery Pro[6], JetBrains AI Refactor, and Copilot Refactor Commands rewrite codebases for better structure, performance, and maintainability.

10. Multi-Agent Systems

OpenAI Reflexion Agents, DevRev AI Engineer Stack[18], and AutoGPT multi-agent frameworks divide software engineering tasks into specialized agent roles.

11. Robotics and Embedded Systems

Skydio AI planners[19], NVIDIA Isaac Robotics GPT[17], and Boston Dynamics AI integrations[20] use LLMs for control code and mission planning.

Conclusion

LLMs are redefining modern software engineering, enabling applications ranging from semantic code search to autonomous program repair. Their adoption is rapidly shaping the future of programming practice and research.

Effectiveness and Impact of Large Language Models in Computer Programming

The integration of Large Language Models (LLMs) into computer programming workflows, programming education, and automated software development pipelines has produced substantial and measurable impacts. A growing body of empirical research demonstrates that LLMs fundamentally reshape how developers learn, generate, debug, and maintain code. Their influence spans three primary domains:

1. **student learning outcomes**
2. **instructional efficiency and curriculum enhancement**
3. **software development performance, accuracy, and reasoning capability.**

Each domain provides strong evidence of the transformative potential of LLMs in both educational and industrial contexts.

1. Impact of LLMs on Student Learning Outcomes

One of the clearest indicators of LLM effectiveness in programming is the significant improvement in student academic performance. According to findings from Exploration of Computer Programming Teaching Reform Based on Large Language Models , students in LLM-supported classrooms achieve substantially better outcomes than those receiving traditional instruction. The study reports that the experimental class using an LLM-integrated teaching system achieved an average score of 85, compared to 76 in the control group. [1] Additionally, the distribution of student performance shifted positively: there were more high-achieving students and significantly fewer low-performing students [2] in the experimental cohort.

These results are notable for two reasons. First, they demonstrate that LLMs help students understand programming concepts more effectively by providing real-time clarification, step-by-step explanations, and immediate error correction. Second, they show that LLMs can serve as adaptive tutors, capable of personalizing instruction based on individual student needs.

The same study highlights that real-time error detection reduces cognitive overload and prevents students from developing misconceptions. The authors write that LLM-based systems "help students find and correct programming errors in time [3]," which improves conceptual understanding and prevents the accumulation of uncorrected mistakes. This immediate feedback loop mirrors one-on-one tutoring, long considered the most effective instructional method in computer science education.

Furthermore, the personalized learning suggestions generated by LLM-driven learning systems enhance student engagement and motivation. The research reports that students received adaptive exercises and recommendations based on their performance, "further improving the learning effect [4]." Personalized learning is a key marker of modern AI-supported pedagogy, and LLMs appear uniquely capable of delivering tailored programming guidance at scale.

2. Impact on Teaching Efficiency and Educational Workflows

LLMs not only benefit students but also significantly reduce instructor workload. One of the most compelling quantitative findings reported in the literature concerns teaching efficiency. In the same study above, researchers document that grading time for programming reports decreased from 150 minutes to only 5 minutes [5], representing a 30× improvement in efficiency. This dramatic time reduction was achieved through automated evaluation, code correctness checks, and feedback generation powered by LLMs.

This efficiency gain has multiple implications:

2.1. Reallocation of Instructional Resources

With routine grading tasks automated, instructors can dedicate more time to:

- course design,
- mentorship,
- curriculum development,
- helping struggling students, and
- preparing more advanced or interactive classroom activities.

This shift from administrative tasks to higher-value pedagogical work directly improves course quality.

2.2. Consistency and Objectivity of Assessment

LLMs produce:

- consistent scoring,
- standardized feedback, and
- unbiased evaluation.

Human grading is subject to fatigue and inconsistency, whereas automated LLM grading follows the same criteria uniformly.

2.3. Enhanced Classroom Participation

The teaching reform study reports that the availability of immediate LLM-based assistance “improves students’ participation [6]”, as they feel more confident seeking help, iterating on their code, and engaging with challenging concepts.

This increase in participation is crucial in programming courses, where students often hesitate to ask instructors for repeated clarifications.

3. Impact on Code Generation Accuracy and Programming Performance

A second major body of research evaluates how effectively LLMs generate code for real programming tasks. The findings from Enhancing Computer Programming Education with LLMs provide extensive quantitative evidence. GPT-4 and GPT-4o consistently outperform open-source models such as LLaMA-3 8B and Mixtral-8x7B across multiple metrics, including correctness, execution time, and code quality.

3.1. High Pass Rates on Programming Tasks

As shown in Table 2 of the study:

- GPT-4 achieves 99% accuracy [8] across most prompting strategies.
- GPT-4o achieves 100% accuracy [8] using the multi-step prompting strategy.

These near-perfect pass rates indicate that state-of-the-art LLMs can reliably generate functionally correct code for a wide range of LeetCode-style tasks.

This demonstrates that LLMs are not merely autocomplete engines—they possess robust problem-solving capability grounded in algorithmic reasoning.

3.2. Code Quality and Maintainability

Pylint scores demonstrate that the code produced by LLMs is not only correct but also structurally sound. GPT-4’s highest Pylint score reached 9.66 [9], indicating strong adherence to Python programming conventions such as naming standards, formatting practices, and modularization. High Pylint scores correlate with:

- code readability,
- maintainability,
- correctness,
- and long-term project sustainability.

This implies that LLMs can generate production-level code, not just rough prototypes.

3.3. Prompt Engineering Effects on Performance

LLM performance improves significantly when aided by structured prompting:

- Base prompts yield 30% success [11] on USACO tasks.
- Multi-step prompts yield 55% success [11].
- Multi-step + specific instructions yield 75% success [11].

These findings show that LLMs can solve far more complex problems—beyond typical training data—when given:

- iterative reasoning steps,
- pseudo-code generation,
- logical verification stages, and
- domain-specific context.

This supports the view that LLMs already possess latent reasoning abilities that can be "unlocked" with carefully engineered input formats.

4. Impact on Complex Problem Solving and Algorithmic Reasoning

Although earlier studies focused on basic programming tasks, recent research demonstrates that LLMs are increasingly effective in advanced, competition-level challenges. The multi-step prompting framework enables the model to break down complex tasks into:

- pseudo-code,
- validation steps,
- test-case analysis,
- edge-case reasoning,
- iterative refinement, and
- final code production.

This approach mimics how expert programmers think, showing that LLMs can serve as scaffolding systems for developing problem-solving skills. The authors emphasize that such strategies “empower LLMs to guide students through complex problem-solving processes [12],” making them useful even for advanced learners preparing for algorithm competitions or high-level computing tasks.

5. Broader Educational and Computational Impact

Across all studies, several broad impacts emerge:

5.1. Increased Accessibility

Students who previously struggled with programming can now receive instantaneous explanations, interactive examples, and personalized debugging support. This democratizes programming education.

5.2. Improved Engagement and Motivation

Immediate feedback increases student confidence, reduces frustration, and encourages experimentation—critical for learning programming effectively.

5.3. Scalability

LLMs allow instructors to support much larger classes without sacrificing personalization.

5.4. Enhanced Professional-Grade Development

Generated code meets professional standards of style and structure, meaning LLMs are increasingly viable in industry settings for rapid prototyping and code augmentation.

Conclusion

Research overwhelmingly supports the conclusion that LLMs produce significant positive impacts in computer programming education and software development. They improve student learning outcomes, dramatically increase teaching efficiency, provide high-quality code generation, and enhance both foundational and advanced problem-solving abilities. Their scalability, adaptability, and precision indicate a transformative shift in the future of programming practice—one in which LLMs serve as both educators and collaborators.

References

[1] https://github.com/

[2] https://copilot.microsoft.com/

[3] https://docs.aws.amazon.com/codewhisperer/

[4] https://www.tabnine.com/

[5] https://www.mintlify.com/

[6] https://www.sourcery.ai/

[7] https://copilot.cloud.microsoft/

[8]
https://replit.com/ai?gad_source=1&gad_campaignid=23286661337&gclid=CjwKCAiA86_J

BhAIEiwA4i9Ju6sRp-tjRGiOwLfnqxpRTX3c8x-cerLDEU8VTbeptAGooiEbDoAeIxoC70sQAvD_BwE

[9] https://www.jetbrains.com/

[10] https://semgrep.dev/

[11] https://snyk.io/product/snyk-code/

[12] https://www.diffblue.com/diffblue-cover/

[13] https://sourcegraph.com/amp

[14] https://www-ibm-com.translate.goog/products/watsonx-code-assistant?_x_tr_sl=en&_x_tr_tl=tr&_x_tr_hl=tr&_x_tr_pto=tc

[15] https://openai.com/tr-TR/index/introducing-codex/

[16] https://projects.eclipse.org/projects/technology.repairnator

[17] https://developer.nvidia.com/isaac

[18] https://devrev.ai/careers

[19] https://www.skydio.com/careers

[20] https://bostondynamics.com/

[21] Y. Wang and X. Chen, “Exploration of Computer Programming Teaching Reform Based on Large Language Models,” pp. 4.

[22] Y. Wang and X. Chen, “Exploration of Computer Programming Teaching Reform Based on Large Language Models,” pp. 4.

[23] Y. Wang and X. Chen, “Exploration of Computer Programming Teaching Reform Based on Large Language Models,” pp. 5.

[24] Y. Wang and X. Chen, “Exploration of Computer Programming Teaching Reform Based on Large Language Models,” pp. 5.

[25] Y. Wang and X. Chen, “Exploration of Computer Programming Teaching Reform Based on Large Language Models,” pp. 4.

[26] Y. Wang and X. Chen, “Exploration of Computer Programming Teaching Reform Based on Large Language Models,” pp. 5.

[27] Y. Wang and X. Chen, “Exploration of Computer Programming Teaching Reform Based on Large Language Models,” pp. 5.

[28] Y. Zhao, “Enhancing Computer Programming Education with LLMs: A Study on Effective Prompt Engineering,” Table 2, p. 9.

[29] Y. Zhao, “Enhancing Computer Programming Education with LLMs: A Study on Effective Prompt Engineering,” Table 4, p. 10.

[30] Y. Zhao, “Enhancing Computer Programming Education with LLMs: A Study on Effective Prompt Engineering,” Table 5, p. 10.

[31] Y. Zhao, “Enhancing Computer Programming Education with LLMs: A Study on Effective Prompt Engineering,” Abstract, p. 1.