Use of Large Language Models in Code Generation

Sanjay Sunil[1], Sudha D[2]

[1]Student, [2]Assistant Professor
[1,2]Department of Computer Applications, SCMS School of Technology and Management, Kochi.

Abstract:
Out of nowhere, big language machines started reshaping how software writes itself, moving far beyond old rigid rules into smart, learning-based engines. These new models - like GPT-4 [1], Claude 3.5 Sonnet [2], DeepSeek-V3 [3], and Gemini Pro [4] - can turn plain speech into working code, jumping between coding styles and tech areas without skipping a beat. Now arriving - LLM-powered coding tools are no longer lab curiosities but live parts of daily developer work. GitHub Copilot, fueled by OpenAI's Codex, sees regular use among coders; research points to faster output under specific conditions. Yet weaving large models into dev pipelines stirs concerns: how solid is the output, can it be trusted, what happens to the coder's craft when machines suggest half the lines. Even with big steps forward, problems remain when using large language models to create code. Though GPT-4 gets functions right most times, its output runs efficiently less than half the time - just 45.4%, while functionally correct in 83.1%. Efficiency lags far behind accuracy, data shows. Code written by machines often clashes in style; research finds mismatches appear in two-thirds to nine-tenths of cases versus human work. Performance shifts wildly based on field - one robot task might succeed 95 out of 100 tries, another drops as low as one in five, depending on which model is picked. This work dives into how well large language models create code, looking at what they can do and where they struggle - covering various systems, coding tongues, settings. Findings from up-to-date research come together here, showing where things stand right now while pointing out gaps that need attention down the road.

Keywords: Large Language Models, Code Generation, Natural Language Processing, LLMs, AI-Assisted Programming

1. INTRODUCTION

Some of the most capable tools available right now — GPT-4, Claude 3.5 Sonnet, DeepSeek-V3, Gemini Pro — don't operate on rigid rule sets. They learn from patterns, and they do it at a scale that wasn't practical even a few years ago. That shift has quietly changed how software gets written. Fixed templates are giving way to adaptive, ML-driven approaches. Instead of predefined logic trees, these systems interpret natural language and generate scripts that hold up across different coding environments. Where older automation tools needed structured, carefully formatted inputs, newer ones draw from massive training datasets to infer intent. The result is that a plain-English request can now produce executable code — and the accuracy keeps improving.

That progress has moved quickly from prototype to practice. What started as experimental integrations are now showing up in real development workflows. GitHub Copilot, built on OpenAI's Codex, spread fast across developer teams — and studies back up what a lot of developers already noticed: it genuinely speeds certain tasks up. That said, leaning on AI assistance raises fair questions. Is the code actually sound? And at what point does convenience start to quietly erode the developer's ownership of what they're building?

The honest answer is that things are still uneven. Research from ENAMEL puts some numbers to it: GPT-4 gets functionality right about 83.1% of the time, which sounds solid until you see that efficiency only holds up 45.4% of the time. Roughly two out of three AI-written code snippets carry styling inconsistencies — small things, but the kind that add up in a real codebase. And performance swings wildly depending on the domain. In robotics alone, results range from around 20% to as high as 95%, depending on the model and the task. That's not a small gap.

What's also notable is how little systematic research has actually examined this. A few patterns are starting to emerge, but the picture is still incomplete. One consistent finding is that performance shifts depending on which programming language is involved — models don't behave uniformly across environments. Work from the past year has highlighted both how capable these tools can be and how unexpectedly they can fall short. Outside of widely-used languages and common use cases, reliability drops noticeably. Progress has been made, but meaningful gaps remain in a lot of test scenarios. For complex, long-running software projects especially, the open questions about long-term dependability haven't been answered yet. The evidence so far suggests that drawing broad conclusions would be premature — more rigorous investigation is still needed before that's warranted.

2. LITERATURE REVIEW

[1] Qiu and his team made ENAMEL with a specific goal in mind: not just to see if AI-generated code works, but also to see how well it works. To do that, they introduced a new metric called eff@k, which builds on the widely used pass@k by factoring in execution speed and working even when runtime data is incomplete. They also raised the evaluation standard by using expert-crafted reference solutions, which consistently outperformed the typical examples found in datasets like HumanEval. When they put 30 different large language models through the framework, the results were telling — not a single one matched the quality of human-written solutions, particularly when the tasks involved complex algorithms or nuanced code edits.

[2] Akter and his team ran a broad comparison between Google's Gemini and OpenAI's GPT models, pulling in ten different benchmarks to cover a wide range of abilities — logical reasoning, factual recall, mathematics, translation, code generation, and instruction following. In standard English tasks, Gemini Pro kept pace with GPT-3.5 Turbo but didn't manage to pull ahead. Where it genuinely stood out was in multilingual translation, especially when converting English into lower-resource languages. That said, it struggled with long numerical sequences and was sensitive to how answer choices were formatted — small things that occasionally shifted its outputs. Even with those quirks, it did a good job of multi-step reasoning tasks, often putting together information better than expected.

[3] Jiang and his team took a broad look at existing research and organized it into four categories: training data preparation, architectural improvements, evaluation methods, and ethical considerations. They mapped out performance timelines and provided concrete benchmark comparisons across HumanEval, MBPP, and BigCodeBench. One of the more striking takeaways was how much the gap between open and closed models has narrowed — DeepSeek-Coder-V2-Instruct and Qwen2.5-Coder-Instruct, for instance, now hold their own against heavier commercial systems. A consistent pattern emerged across model variants: instruction-tuned versions consistently outperformed their base counterparts.

[4] DeepSeek-AI's technical report talked about something important: DeepSeek-V3, a huge language model with 671 billion parameters that is based on a Mixture-of-Experts architecture. The scale sounds overwhelming, but in practice only 37 billion parameters are activated per token, which keeps the computational load much leaner than the raw number suggests. Rather than relying on continuous fine-tuning signals, the model was pre-trained on 14.8 trillion tokens — enough data to internalize patterns broadly and deeply. Its Multi-head Latent Attention mechanism moves away from conventional attention techniques, prioritizing faster output generation instead. Load balancing is handled efficiently as well, notably without the auxiliary loss functions that most systems of this size depend on. Benchmark results were strong — DeepSeek-V3 outperformed established open-source models and sat comfortably alongside top proprietary systems, with coding tasks, numerical reasoning, and logical problem-solving being particular strengths. Perhaps just as notable was how smoothly training went: the entire process completed in roughly 2.788 million H800 GPU hours, a figure that came in lower than most would have predicted for a model of this scale.

[5] Bui and his team developed OPENIA to address something most evaluation tools miss entirely — the reasoning process behind AI-generated code, not just the final output. By examining intermediate internal states within Code LLMs like DeepSeek-Coder, CodeLlama, and MagicCoder as they construct solutions, the

system picks up on hidden patterns that frequently indicate whether an answer will hold up before it's even finished generating. The results from testing were hard to ignore: OPENIA outperformed existing tools by a considerable margin, doubling success rates on general coding tasks and cutting error rates nearly in half on repository-level projects. For developers, that kind of early signal matters — catching a flaw in how the model is reasoning is a lot more useful than discovering a bug only after the code has already been written.

[6] Wang and colleagues were curious about something specific — not whether AI can write code, but how its code actually differs from what a human would write. They picked five widely used models and ran direct comparisons, looking at differences across five areas: how code is structured, how intent shifts, patterns in language use, the reasoning paths taken, and how errors get handled.The results were surprising because the biggest differences weren't in big architectural choices, but in small things like spacing habits and short snippets. The AI-generated code was mostly coherent, neat, and reliable enough to follow, even with those quirks. Tweaking prompts helped fix mistakes sometimes, but pushing for more precision occasionally backfired — the responses got longer and, ironically, harder to use.

[7] First up, Zheng's team studied AI models designed only for programming - so-called Code LLMs. Not limited to surface traits, these tools were tested thoroughly against wider, all-around counterparts. The main curiosity driving things: can narrow designs outperform broad ones? As findings piled up, certain trends appeared - not everywhere did focus bring clear gains

One model might manage specific programming tasks more effectively than another. Similar in size, those built for coding usually win over general-purpose versions. Consider CodeFuse-CodeLlama-34B - it did better at creating code compared to wider models. Still, something like GPT-4 stayed close behind across various software work. Higher scores appeared once models focused on specific jobs. Without that focus, big size still fell short.

[8] A fresh look at how ChatGPT-4o writes code in Python versus Java came from work led by Almanasra and Suwais, who ran a side-by-side review Three hundred challenges in data structures, alongside three hundred from LeetCode. Though Java-built solutions ran faster, especially on tougher tasks, Python used less memory no matter how complex the problem got. Code written by artificial intelligence had flaws - poor comments and weak error checks appeared often in both tongues. The model did well when the problems were simple, but it quickly showed its limits when they got harder. Even if you started off strong, performance went down as the difficulty went up. What works just doesn't work when things get tough.

[9] One test by Jiang and team checked how well DeepSeek, ChatGPT, and Claude handled science math jobs. While tackling tricky number puzzles like PDEs, each model had to make complex choices. Instead of standard setups, those built for logic steps did much better at tough questions. When speed mattered, especially in thinking through answers, ChatGPT 03-mini-high moved quicker than others. Performance gaps stood clear across all trial runs.

[10] Sobo and team kicked things off by pitting three models - ChatGPT 3.5, Gemini 1.5 Pro, and Claude 3.5 Sonnet - against each other in coding tasks tied to human-robot interaction. Twenty different prompts were used; these started basic, like moving a robot arm, then stepped up to tricky actions such as handling objects. Results weren't close - Claude hit 95% success, while Gemini landed at 60%, ChatGPT trailed behind with just 20%. Where ChatGPT stumbled most? Ignoring key parts of the given directions made a big difference.Twelve out of sixteen times it didn't work - proof that even though large language models might help robots, they still struggle with understanding space and following limits.

[11] Experiments by Poldrack and team put GPT-4 through its paces on AI-supported programming tasks. Despite strong performance in generating code, close oversight by people stays essential for correctness. Refactoring old code? The model made noticeable upgrades when judged against standard benchmarks. When creating test cases, it reached wide coverage - yet most failed outright, needing manual fixes. Mistakes popped up often with math-heavy sections, where the system invented flawed equations out of nowhere. Because of

this tendency, final approval should always rest with a person watching closely. Results only hold if someone checks each step carefully after the machine does its part.

[12] On a particular day, Nejjar and his team looked closely at how big language models help scientists write code for their research. They didn't just talk about it; they actually tried it out with real tasks, like drawing graphs or doing math on multiple threads. ChatGPT, Bard by Google, and Copilot by GitHub were some of the tools that were looked at. Some outputs were smooth, clean, and correct, while others told stories that sounded like logic but weren't. The numbers looked right but weren't. Mistakes snuck in silently, pretending to be facts, which made false confidence grow without anyone noticing. The researchers call this tendency to make up answers while sounding sure confabulation. When machines guess instead of know, they make mistakes that change the results without any clear warning signs. Efficiency may increase temporarily, although confidence in results may diminish subtly. When sources mix up creation and recall, truth becomes weak. Things seem to be moving quickly until someone checks the base.

[13] Anthropic unleashed three models, Opus, Sonnet and Haiku, halfway into its newest release, which are not only faster but also more precise at logic tasks. These systems work with text and images, a step forward in their understanding of real-world inputs. Opus beat out scores in knowledge quizzes and deep science questions during tests. Exposure to the world conversation till mid-2023 means each version speaks several tongues better than before. Training utilized Constitutional AI, modifying responses to common principles. These are less evasive than previous builds when asked normal things. Fluency is growing everywhere, without resorting to artificial phrasing. Benchmarks change because performance is now a matter of subtle judgment calls.

[14] Chen's team studied language models that were built for coding tasks. Then came HumanEval – a set of 164 custom programming challenges designed to check whether the generated code actually works, using automated tests. They judged success not on one guess, but on whether any guess out of k passes. This was the start of pass@k. Things started to fall into patterns from there -- Codex got more correct when it tried multiple times. In persistence, not perfection, grew success, each try another chance. The thing that jumped out was the ways to boost outcomes by just generating more solutions.

3. ALGORITHMS AND ARCHITECTURES

Most current coding models rely on Transformers, yet their learning methods differ widely. Because of how they handle data during training, some use one path forward, others split tasks across many paths. Depending on design - decoder alone, paired encoder-decoder, or MoE - the way decisions unfold changes sharply. Each structure steers computation differently when generating output. Performance shifts not just by size, but by these hidden routes inside.

3.1 ChatGPT (GPT-3.5/4)

The models that power the chat bot are based on a decoder-only Transformer architecture and use a combination of established algorithms [3].

- Core Training: Autoregressive Causal Language Modeling (CLM), primarily next token prediction but a lot of instruction. Tuning to follow what the user says [11] .
- Alignment: Employs Reinforcement Learning from Human Feedback (RLHF) for aligning model outputs with human intention of being helpful and correct. [2].
- Inference: Generates diverse code outputs via in-context learning (few-shot and zero-shot) and decoding techniques like greedy search, beam search, and sampling. [8].

3.2 Claude 3

Claude 3 models also use a multimodal, decoder-only Transformer architecture, but are different because of their own alignment method. [13].

- Core Training: It begins with pre-training in an unsupervised way (CLM) and is further improved by instruction tuning [13].

- Alignment: The main feature is Constitutional AI, a kind of reinforcement learning where the model is aligned to a set of simple rules ("a constitution") to ensure that it is safe and helpful [13].
- Inference: Well suited for Chain-of-Thought (CoT) prompting for multi-step reasoning and fine-tuned for effective tool use (function calling) [10].

3.3 Gemini

Google's Gemini is a decoder-only Transformer model that was trained on a wide range of data types from the start [2].

- Core Training: Unlike most approaches that stick to a single modality, it uses autoregressive language modeling pre-trained across text, images, and video together — a broader foundation than many of its counterparts [2].
- Alignment and Inference: Instruction fine-tuning and RLHF are used to refine behavior post-training, consistent with standard alignment practices. During inference, it leans heavily on Chain-of-Thought prompting, which is where much of its reasoning capability actually shows up [2].

3.4 DeepSeek-V3

DeepSeek-V3 is a capable open-source model built on a Mixture-of-Experts (MoE) Transformer architecture, a design choice that directly contributes to its efficiency and ability to scale [4].

- Architecture Innovations: It introduces Multi-Head Latent Attention (MLA) to reduce memory pressure on the KV cache, paired with a DeepSeekMoE architecture that handles load balancing without auxiliary loss — keeping training more stable and noticeably faster [4].
- Core Training: Multi-Token Prediction (MTP) sits at the heart of its training approach. Rather than predicting one token at a time, the model learns to anticipate several future tokens simultaneously, which makes speculative decoding considerably more efficient [4].
- Alignment: Post-training involves a combination of Supervised Fine-Tuning (SFT), RLHF, and knowledge distillation drawn from specialized reasoning models — each stage sharpening the model's real-world usability [4].

4. HUMANEVAL PERFORMANCE COMPARISON

The comprehensive analysis of the research literature [1], [2], [4], [10] has led to the creation of Table I, which shows the HumanEval performance scores for the most important LLMs in code generation.

TABLE I-CORE HUMANEVAL PERFORMANCE COMPARISON [1], [2], [4]

	HumanEval Pass@1	**Notes**
Claude 3.5 Sonnet	84.9%	Leading performance
GPT-4 Turbo	67.0%	Strong general-purpose
DeepSeek-V3	65.2%	Cost-effective
Gemini 1.5 Pro	71.9%	Multimodal approach
Code Llama	43.3%	Specialized code model
StarCoder	53.0%	Open-source

The ENAMEL benchmark [1] shows a very important finding, as seen in Table ??: even GPT-4 only works 45.4% of the time (eff@1), even though it is 83.1% correct (pass@1). This big difference shows that LLMs can make code that works, but they often make implementations that aren't very efficient algorithmically.

TABLE II- EXTENDED HUMANEVAL ANALYSIS WITH EFFICIENCY AND EFFICIENCY METRICS

	Pass@1	**Eff@1**	**Style**
GPT-4	83.1%	45.4%	66-90% inconsistent
Claude 3.5	95%*	N/S	Better than average

	Pass@1	Eff@1	Style
DeepSeek-V3	65.2%	N/S	Variable
Gemini Pro	65.22%	N/S	60% success rate

*95%success rate specifically for HRI tasks [10]; N/S = Not Specified

Table III shows that the evaluation of LLMs for Human-Robot Interaction tasks [10] shows huge differences in performance. Claude 3.5 Sonnet has a success rate of 95%, which is much better than Gemini 1.5 Pro (60%) and ChatGPT 3.5 (20%). This difference shows how important it is to choose the right model for specific uses.

TABLE III- DOMAIN-SPECIFIC PERFORMANCE COMPARISON [8]–[10]

	ChatGPT	Gemini	Claude
Human-Robot Interaction	20%	60%	95%
Scientific Computing	Variable	Variable	Variable
Code Efficiency	45.4%**	N/S	N/S
Cross-Language Tasks	Good	Excellent	Good

**GPT-4 efficiency score [1]; N/S = Not Specified

4.1 Key Insights from Performance Analysis

You can see a few important things by looking at Tables I, II, and III all at once. What do you notice? A big difference between code that works and code that runs well [1]. Not every right answer makes good use of resources. When you use software for real, how fast it works is just as important as whether it works at all. That difference affects what happens after the launch.

A study found that the code that big language models make doesn't usually match the styles that people use when they write programs. About two-thirds to more than nine out of ten examples show some kind of difference. A classification system lists about two dozen different types of mismatches. There are five groups of these: how lines are spaced and shaped, how meaning changes, how statements or expressions differ, how logic paths are not always followed correctly, and how errors are not always handled well.

Domain Specialization Patterns: Various models exhibit unique advantages within specific application domains [9], [10]:

- Claude 3.5 Sonnet: Works best when safety is a priority — it also handles spatial reasoning better than most others in its class.
- GPT-4 models: Hard to beat for general use. It covers a wide range of tasks and rarely struggles when the reasoning gets complex.
- DeepSeek-V3: Punches above its weight in math, and what makes it stand out further is that it gets there without the hefty training costs most comparable models carry.
- Gemini models: Genuinely impressive with languages — switching between them, including less common ones, is where it shines. Math reasoning, though, is still a weak spot.

Marks for surprisingly low spending DeepSeek-V3's best results cost only $5.576 million [4]. That number suggests something bigger: strong models don't have to use up resources. Performance goes up, but computing needs stay the same. Breakthroughs now happen with less money. Efficiency changes what can be done. Not spending as much doesn't mean you're falling behind.

Something worth noting when models switch between programming languages — the differences aren't surprising once you've seen enough of them. Java moves faster through tough problems, Python quietly uses less memory. Neither is better overall; they just behave differently, and that behavior traces back to what each model was exposed to during training.

But here's the thing developers often miss: whether the code runs isn't really the finish line. The teams getting the most out of these tools aren't just reviewing outputs — they're paying attention to how the whole process feels while they're building. Oddly enough, the models that get adopted aren't always the most accurate ones. They're the ones that fit naturally into how someone already works. The editing back-and-forth matters. The small adjustments, the prompt tweaks, the moments where something almost works — those shape the experience more than any benchmark does. And some problems don't show up in testing at all; they only surface once someone's actually using the thing day-to-day. Getting that integration right turns out to matter just as much as anything the model produces on its own.

5. CURRENT STATE OF LLM CODE GENERATION

5.1 Model Performance Landscape

Current LLM code generation capabilities demonstrate remarkable progress, with leading models achieving high functional correctness rates across standard benchmarks. The performance hierarchy shows clear distinctions:

1) Claude 3.5 Sonnet: Excellent for safety-critical applications with 95% success in HRI tasks [10]
2) GPT-4 models: Strong across diverse applications with 83.1% pass@1 on HumanEval [1]
3) DeepSeek-V3: Cost-effective with competitive performance at only $5.576M training cost [4]
4) Gemini models: Strong multilingual capabilities but challenges in mathematical reasoning [2]

5.2 Quality Beyond Correctness

Even though code works, the structure of AI-generated outputs is often not very good these days. When human programmers write code, they try to strike a balance between logic and clarity. When machines write code, they tend to make it messy with clunky loops and strange naming choices. Even when the end result looks good on the surface, efficiency drops. Testing with ENAMEL shows that GPT-4 can only reach about half of the best performance markers, but it still gets the job done. That mismatch is most important when software is put into real-world systems that need to be kept clean and run quickly.

5.3 Domain-Specific Capabilities

Some models do a good job with everyday coding, but they have trouble with niche areas that require expert knowledge. Success rates vary greatly from field to field; for example, in robotics, they range from 20% to 95% [10]. Even though there are a lot of general skills, focused training might be the most important when accuracy is important. What works for most people may not work for everyone.

6. KEY FINDINGS AND IMPLICATIONS

6.1 Performance Hierarchy and Specialization

When you look at different research comparisons, you can see how some models do better than others depending on the job they have to do.

- Claude 3.5 Sonnet: Better for applications where safety is important and for spatial reasoning
- GPT-4 models: Great at a lot of things and very good at reasoning
- DeepSeek-V3: Affordable training with good math skills that are competitive
- Gemini models: Can work well in many languages, but math thinking fights back

6.2 Critical Limitations

Current LLM code generation systems have a number of persistent limitations:

1) The Efficiency-Correctness Gap: There is a big difference between generating code that is correct and code that is efficient [1].
2) Style Differences: 66–90% of the code that was generated has a different style than human code [6].
3) Domain Expertise Requirements: Human oversight is still needed for validation
4) Context Sensitivity: Performance drops in complicated situations that depend on context

6.3 Evaluation Framework Evolution

Things are changing in how we check performance in the area now. People used to only care about whether something was right or wrong. Today, there are new ways to look at overall quality. Efficiency checks have begun to emerge [1], providing new insights. Along with them, there has been a greater focus on keeping the tone and structure the same [6]. These steps make broader review strategies that are slowly coming together more important.

7. FUTURE DIRECTIONS

7.1 Technical Improvements

In the future, studies should focus on these areas.

- Training for Efficiency: Making models that work well and are correct at the same time
- Style-Consistent Generation: Making it easier to follow coding standards and best practices
- Fine-tuning for specific domains: Making models work better for certain uses. Quality checks happen right away, using built-in model signals to guess how reliable the output is [5].

7.2 Evaluation Framework Development

- Full set of quality metrics: going beyond just functional correctness to include maintainability and efficiency
- Domain-Specific Benchmarks: Special tests for different types of applications
- Long-term Code Evolution: Checking how the code that was made works over time

7.3 Practical Deployment

- Working together with AI: Finding the right balance between automation and human oversight
- Context-Aware Generation: Making it easier to understand what the project needs
- Quality Assurance Integration: Adding the ability to evaluate to development tools

8. CONCLUSION

It almost caught everyone off guard — AI systems that can actually write code, and write it well enough to pass most benchmark tests. The impressive part isn't hard to see. The harder part is what happens after: taking that code out of a controlled test and dropping it into a real software project, where the stakes are different and the margin for error is a lot smaller.

What research keeps pointing to, though, is a consistency problem. The code isn't broken — it just isn't good. Models like GPT-4 and Claude 3.5 Sonnet can usually follow the logic, but the output they produce in real conditions tends to be cluttered, difficult to read, and occasionally just wrong. Performance dips. The style shifts unpredictably. And the responses, while often close, aren't reliable enough to trust without review.

Specialized domains make these limitations even harder to ignore. In robotics, there's measurable progress — that much is clear. But introduce complex mathematics into the equation and the cracks show up fast. Domain knowledge still matters enormously, and human judgment hasn't been replaced — it's still driving the whole process. A solution that holds up in one environment often falls apart in another.

Evaluation has improved considerably over the last few years, which is part of why we even know this. Modern benchmarking tools can now measure speed and reliability at the same time, making it much easier to assess what large models genuinely deliver versus what they just appear to.

So will LLM-based coding tools actually work long-term? Probably — but not without some things falling into place first. Better performance monitoring, more task-specific architectures, and workflows built around real human-AI collaboration are all part of what that future needs. The pieces are starting to align. But closing the gap between a model that performs well in testing and one that holds up in production requires active effort. That gap won't close on its own.

REFERENCES:

1. R. Qiu, W. W. Zeng, J. Ezick, C. Lott, and H. Tong, "How Efficient is LLM-Generated Code? A Rigorous & High-Standard Benchmark," ICLR 2025, 2025.
2. S. N. Akter, Z. Yu, A. Muhamed, T. Ou, A. Ba¨uerle, A´ . A. Cabrera, K. Dholakia, C. Xiong, and G. Neubig, "An In-depth Look at Gemini's Language Abilities," arXiv preprint arXiv:2312.11444v2, 2023.
3. J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A Survey on Large Language Models for Code Generation," ACM Trans. Softw. Eng. Methodol., vol. 37, no. 4, 2024.
4. DeepSeek-AI, "DeepSeek-V3 Technical Report," arXiv preprint arXiv:2412.19437v2, 2024.
5. T.-D. Bui, T. T. Vu, T.-T. Nguyen, S. Nguyen, and H. D. Vo, "Correctness Assessment of Code Generated by Large Language Models Using Internal Representations," arXiv preprint arXiv:2501.12934v1, 2025.
6. Y. Wang, T. Jiang, M. Liu, J. Chen, M. Mao, X. Liu, Y. Ma, and Z. Zheng, "Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models," Proc. ACM Softw. Eng., vol. 2, no. FSE, Article FSE032, 2025.
7. Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, "A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends," ACM Trans. Softw. Eng. Methodol., vol. 1, no. 1, 2024.
8. S. Almanasra and K. Suwais, "Analysis of ChatGPT- Generated Codes Across Multiple Programming Languages," IEEE Access, vol. 13, pp. 23580-23596, 2025.
9. Q. Jiang, Z. Gao, and G. E. Karniadakis, "DeepSeek vs. ChatGPT vs. Claude: A comparative study for scientific computing and scientific machine learning tasks," Theoretical and Applied Mechanics Letters, vol. 15, Article 100583, 2025.
10. A. Sobo, A. Mubarak, A. Baimagambetov, and N. Polatidis, "Evaluating LLMs for Code Generation in HRI: A Comparative Study of ChatGPT, Gemini, and Claude," Applied Artificial Intelligence, vol. 39, no. 1, e2439610, 2025.
11. R. A. Poldrack, T. Lu, and G. Beguš, "AI-Assisted Coding: Experiments with GPT-4," arXiv preprint arXiv:2304.13187v1, 2023.
12. M. Nejjar, L. Zacharias, F. Stiehle, and I. Weber, "LLMs for science: Usage for code generation and data analysis," Journal of Software: Evolution and Process, vol. 37, e2723, 2025.
13. Anthropic, "The Claude 3 Model Family: Opus, Sonnet, Haiku," Model Card, 2024.
14. M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power,L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings,M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.