The Future of Programming: Exploring a Bytecode Intermediary Optimised by Large Language Models

1. Introduction: The Evolving Landscape of Programming with LLMs

The field of software engineering is undergoing a significant transformation with the increasing integration of Large Language Models (LLMs) into various aspects of the development lifecycle. Studies have consistently demonstrated the remarkable progress of LLMs, particularly in their capacity for code generation.[1] These models are now being employed to assist developers in a wide array of tasks, ranging from the mundane completion of code snippets to the more complex generation of entire code blocks based on natural language instructions.[3] The initial success of LLMs in these areas suggests a fundamental shift in how software might be created in the future, yet the current model of direct interaction with human-constructed programming languages may not be the most effective way to fully leverage the potential of these advanced AI systems.

Current programming languages, characterised by their specific syntax and semantic rules, are primarily designed to facilitate human comprehension and logical expression. However, these languages might not represent the most efficient interface for LLMs, especially when considering the need for deep optimisation and interaction with hardware at a fundamental level.[7] This creates a potential disparity between the high-level abstractions that make programming accessible to humans and the low-level understanding that LLMs could potentially achieve if provided with a more suitable intermediate representation.

What can we do in a situation like this? Could we introduce a bytecode that acts as a central intermediary in the programming process? This bytecode would serve as a *common language*, bridging the gap between human-facing tools, which could potentially be closer to natural language, and LLM-based optimizers that possess a more profound understanding of the underlying hardware. Furthermore, LLMs themselves could be instrumental in generating this bytecode, mirroring their current capabilities in handling human language. The control of the exact bytecode would naturally be in the hands of humans still. This report aims to explore the fundamental rationale behind this concept, to thoroughly examine the arguments both in favour and against its realisation based on current research and the understanding of LLMs and compiler design, and ultimately to discuss the potential future of programming within such a landscape.

2. The Rationale for a Bytecode Intermediary

LLMs are exhibiting increasingly sophisticated natural language processing capabilities.[1] Their ability to understand context, nuance, and even generate creative text is constantly improving. However, programming languages, while possessing a defined grammar, are constructed for human readability and logical articulation, which may not perfectly align with the way LLMs internally process information.[7] The core strength of LLMs lies in their capacity to process and generate structures that resemble language. If the representation of programs could be made more inherently "understandable" to LLMs, it might unlock unprecedented levels of efficiency and optimisation. Human languages often contain ambiguities and redundancies that LLMs are trained to resolve. Programming languages, conversely, strive for precision and typically lack such redundancies. A bytecode representation could potentially offer a middle ground, providing a structured format that LLMs can process effectively without the complexities associated with high-level language abstractions.

Human programmers dedicate significant time and effort to optimising code for performance, taking into account factors such as algorithmic efficiency, data structure choices, and the inherent limitations of the target hardware. LLMs, with their inherent ability to analyse vast quantities of data and discern complex patterns, could potentially surpass human optimisation capabilities if they were able to directly manipulate a lower-level representation like bytecode.[11] In this scenario, LLMs could function as highly intelligent compilers, moving beyond the traditional static analysis and rule-based optimisations to leverage their learned understanding of both code and hardware. Traditional compilers operate based on a predefined set of rules. LLMs, having been trained on extensive datasets of both optimised and unoptimised code, and potentially on hardware performance data, could learn to identify more intricate and context-aware optimisation strategies. This could lead to the generation of code that is considerably more efficient than code optimized by human experts in many instances.

While the current generation of LLMs primarily operates on textual data, there exists a significant potential to train them on data pertaining to hardware architectures, performance metrics, and the effects of different bytecode instructions on various hardware platforms.[14] This training could enable LLMs to generate or optimise bytecode that is specifically tailored for the underlying hardware, potentially leading to substantial gains in performance.[16] By effectively bridging the gap between the software representation in the form of bytecode and the understanding of hardware through specialized training data, LLMs could become pivotal in achieving optimal

utilisation of hardware resources.

3. Arguments in Favor of a Bytecode-Centric Future

LLMs possess the capability to analyse bytecode and identify opportunities for optimisation that might be overlooked by human programmers or even traditional compilers. This could encompass a range of low-level optimisations, including the reordering of instructions, the allocation of registers, and other hardware-specific adjustments tailored to particular architectural nuances.[16] LLMs could introduce a novel form of "learned" optimisation, moving beyond the limitations of rule-based approaches employed by current compilers. Having been trained on extensive datasets of both optimised and unoptimised code, these models could potentially discern more subtle and effective optimisation strategies. The growing body of research focused on LLM-based compiler optimisation [13] supports the feasibility of this direction.

In such a future, developers might interact with programming through more intuitive interfaces, possibly utilising commands that resemble natural language or specialised languages designed for specific domains. These high-level descriptions would then undergo translation into the intermediary bytecode, allowing humans to concentrate on the core logic and functionality of their programs without needing to be concerned with intricate low-level details or the complexities of optimisation.[3] This could potentially democratise programming, making it more accessible to individuals who do not possess expertise in traditional programming languages. The concept of programming using natural language has been a long-term aspiration. LLMs, with their advanced natural language processing capabilities, could potentially realise this vision by acting as intelligent intermediaries, translating from natural language or high-level intent to a more machine-friendly bytecode format. Research efforts exploring the use of natural language for code generation and interaction provide support for this possibility.[3]

A universal bytecode could potentially execute on any hardware platform that has a compatible virtual machine, similar to the way Java bytecode operates.[27] LLMs could then optimise this bytecode specifically for the hardware it is running on, ensuring both broad portability and tailored performance.[52] This approach could significantly simplify the distribution and deployment of software across a diverse range of hardware environments. The concept of using bytecode for achieving platform independence is well-established. Extending this with LLM-driven optimisation for specific hardware could potentially offer the advantages of both: the "write once, run anywhere" capability coupled with performance that is specifically tuned to the

execution environment. Projects such as the Meta LLM Compiler, which operates on LLVM IR (a form of bytecode), indicate the potential of LLMs in this area.[13]

The separation between human-facing languages and LLM-optimized bytecode could potentially pave the way for the development of entirely new programming paradigms that are better suited for both human expression and AI-driven optimisation. This could lead to the creation of more efficient and powerful methods for developing software, methods that we may not have even imagined yet. Current programming paradigms are largely shaped by the limitations and strengths of human cognition. An AI-centric intermediary could facilitate paradigms that leverage the unique capabilities of LLMs, possibly leading to more declarative or goal-oriented programming approaches.

4. Challenges and Arguments Against a Bytecode-Centric Future

Designing a bytecode that is sufficiently expressive to represent the full complexity of modern programming languages, while simultaneously being efficient for manipulation by LLMs and execution on diverse hardware, would present a formidable challenge.[51] Achieving the optimal balance between expressiveness and efficiency for both human interaction (indirectly) and LLM processing within a single bytecode could prove to be a difficult undertaking. Existing bytecodes, such as Java bytecode or.NET CIL, are already quite intricate. The development of a new bytecode that specifically caters to the needs of LLM-based optimization while also serving as a viable compilation target for a wide range of high-level languages would necessitate careful consideration of instruction sets, data types, and control flow mechanisms. The inherent complexity of existing intermediate representations [51] underscores the magnitude of this challenge.

Debugging and maintaining code that is primarily represented in bytecode could pose significant difficulties for human developers, particularly if the original high-level source code is substantially different or if the optimisations performed by the LLM introduce unexpected behavior.[62] The potential loss of direct human readability and control over the optimised code could impede efforts in debugging and maintenance. While LLMs might be able to provide assistance with debugging at the bytecode level, human developers could find it challenging to reason about and trace the execution flow within a low-level representation, especially if they were not the original authors. The development of specialised tools and techniques for debugging and maintaining bytecode-centric systems would be essential.[63]

Entrusting LLMs with the generation and optimisation of code at a low level like

bytecode could introduce novel security vulnerabilities if these LLMs are compromised or make incorrect optimization decisions that inadvertently create security flaws.[62] Ensuring the security and reliability of bytecode generated and optimised by LLMs would be of paramount importance. LLMs have been observed to sometimes generate code that contains security vulnerabilities.[68] If they are responsible for generating and optimising bytecode, there is a potential risk of introducing low-level vulnerabilities that might be more difficult to detect and subsequently exploit. Robust verification and validation mechanisms would therefore be necessary.[76]

The future role of existing high-level programming languages in such a scenario remains uncertain. It is possible that they might not become entirely obsolete. Instead, they could continue to serve as a more natural and expressive means for humans to define complex program logic, with the compilation to bytecode becoming an intermediate step in the overall process.[78] Existing programming languages might evolve to become more conducive to AI processing or find specialised niches within this new ecosystem. While a bytecode intermediary might emerge as a central component, the need for programming languages that are both readable and writable by humans is likely to persist, at least in the near future. These languages might adapt to better facilitate their translation into an AI-optimized bytecode.

Introducing an additional layer of bytecode and the associated translation processes between human-facing tools, the bytecode itself, and the final machine code could potentially introduce performance overhead.[35] While the primary goal of LLM optimisation would be to enhance performance, the initial translation steps and the execution of bytecode within a virtual machine could incur certain costs. The ultimate performance benefit would depend on the efficiency of these translation processes and the effectiveness of the optimisations performed by the LLM. Although bytecode offers the advantage of portability, it typically requires a virtual machine or interpreter for its execution, which can introduce overhead compared to the direct execution of native machine code.[47] Techniques like just-in-time compilation can help to mitigate this overhead, but the balance between any introduced overhead and the performance gains achieved through LLM optimisation would need careful management. Research into the efficiency of code generated by LLMs [86] suggests that current models do not always produce the most performant code.

5. The Future of Programming: A Two-Sided Coin

In the future, programming interfaces designed for human interaction could become significantly more abstract and intuitive, potentially leveraging advanced natural

language processing to understand and translate high-level intentions into bytecode.[3] Domain-specific languages, tailored to address the specific needs of particular tasks or industries, could also become more prevalent, offering a more effective balance between human expressiveness and the ease with which they can be translated into bytecode.[90] Programming could become more accessible to a wider range of individuals, reducing the current reliance on memorising complex syntax and understanding intricate low-level details. The ongoing trend toward higher levels of abstraction in programming is likely to continue, and LLMs could play a crucial role in enabling more natural and intuitive ways for humans to articulate their computational requirements.

LLMs themselves would likely evolve into highly sophisticated optimisers, possessing a deep understanding of both the logical structure of software, as represented in bytecode, and the underlying architectures of various hardware platforms.[14] These AI systems could continuously analyse and refine bytecode to achieve optimal performance, dynamically adapting to different hardware environments and evolving performance demands.[16] Artificial intelligence could become an indispensable component of the compilation and execution pipeline, ensuring that software operates as efficiently as possible on any given hardware. The increasing capabilities of LLMs in code-related tasks, including optimisation [13], suggest a future where AI assumes a far more active and central role throughout the entire software lifecycle, including the critical aspect of performance tuning.

The bytecode would serve as the essential intermediary, a structured representation capable of being effectively generated from human input and efficiently processed and optimized by LLMs for execution on hardware.[27] Its design would necessitate a careful balancing of the needs of human expression (albeit indirectly) and the processing requirements of AI systems.[51] The design of this future bytecode would be a critical determinant of the success of this emerging programming paradigm. The bytecode would need to be sufficiently expressive to capture the full spectrum of computational tasks, structured in a way that allows LLMs to effectively analyse and optimise it, and readily translatable into efficient machine code for a diverse range of hardware architectures.

Looking further ahead, LLMs might even develop the capability to directly generate bytecode from high-level descriptions or natural language, potentially bypassing the need for traditional compilers in certain scenarios.[93] This would require LLMs to possess a very deep understanding of both programming semantics and the intricacies of bytecode instructions.[94] Such a development could further streamline the programming process, with AI assuming an even more central and direct role in

the creation of software. As LLMs continue to improve their proficiency in understanding and generating code, their ability to directly produce lower-level representations like bytecode appears increasingly plausible. This would likely involve training LLMs on large and diverse datasets consisting of natural language descriptions paired with their corresponding bytecode representations. Projects like ByteCodeLLM, which utilizes LLMs to decompile bytecode back into source code [94], already demonstrate that LLMs can learn the complex relationships between high-level code and its bytecode equivalent.

6. Conclusion: Reimagining the Programming Landscape

The concept of partitioning the human aspects of programming through a bytecode intermediary that is intelligently optimised by LLMs presents a compelling and thought-provoking vision for the future of software development. The potential benefits of such an approach are numerous and significant. These include the possibility of achieving enhanced optimization levels that surpass human capabilities, providing greater abstraction for human programmers and potentially democratising software creation, enabling improved portability and interoperability across diverse hardware platforms, and even fostering the emergence of entirely new programming paradigms. Current research on LLM-driven compiler optimisation and the advancements in natural language programming provide a degree of credibility to these potential outcomes.[3]

However, the realisation of this vision is not without considerable challenges. Significant hurdles remain in areas such as the complex design of a bytecode that effectively serves both human intent and AI processing, addressing potential difficulties in debugging and maintaining code primarily represented in bytecode, mitigating security risks associated with AI-driven code generation and optimisation at a low level, understanding the evolving role of existing programming languages in such a future, and carefully managing potential performance trade-offs introduced by the additional layer of abstraction.[35]

The future of programming in this context will likely involve a more tightly integrated collaboration between humans and artificial intelligence. Humans could increasingly focus on defining the desired outcomes and high-level logic of their software, while LLMs take on a greater responsibility for generating and optimising the underlying code through a carefully designed bytecode intermediary. This division of labor could potentially lead to more efficient, portable, and powerful software systems.

To fully realise this potential, further research is essential. Key areas of focus should

include exploring the optimal design principles for such a bytecode, developing robust tools and techniques for debugging and maintaining systems built around this bytecode representation, and establishing effective mechanisms to ensure the security and reliability of code that is both generated and optimised by LLM systems. The growing interest within the research community in the intersection of LLMs and intermediate representations for software development [19] strongly suggests that this area will continue to be a significant focus of innovation and exploration in the years to come.

Feature	Traditional Programming	Bytecode-Centric Programming with LLMs
Human Interface	Programming languages (Python, Java, C++, etc.)	Potentially natural language, DSLs, high-level tools
Optimisation	Primarily manual, relies on developer expertise	LLM-driven, hardware-aware optimisation of bytecode
Portability	Achieved through language-specific VMs or compilation	Inherently portable bytecode with LLM-based adaptation for specific hardware
Hardware Understanding	Indirect, through developer knowledge and compiler hints	Direct, through LLM training on hardware data
Code Representation	High-level source code	Intermediary bytecode
AI Involvement	Primarily in code generation assistance	Central role in optimisation and potentially direct bytecode generation

Works cited

1. Evaluating the Generalization Capabilities of Large Language Models on Code Reasoning, accessed on May 18, 2025, https://arxiv.org/html/2504.05518v1
2. Paper2Code: Automating Code Generation from Scientific Papers in Machine Learning - arXiv, accessed on May 18, 2025, https://arxiv.org/pdf/2504.17192
3. Prompting LLMs for Code Editing: Struggles and Remedies - arXiv, accessed on May 18, 2025, https://arxiv.org/html/2504.20196v1
4. arXiv:2407.12504v2 [cs.CL] 8 Feb 2025, accessed on May 18, 2025, https://arxiv.org/pdf/2407.12504
5. A Survey on Large Language Models for Code Generation - arXiv, accessed on

May 18, 2025, https://arxiv.org/html/2406.00515v2
6. Introducing Code Llama, a state-of-the-art large language model for coding - Meta AI, accessed on May 18, 2025, https://ai.meta.com/blog/code-llama-large-language-model-coding/
7. The art of programming and why I won't use LLM - Hacker News, accessed on May 18, 2025, https://news.ycombinator.com/item?id=41349443
8. Shifting Long-Context LLMs Research from Input to Output - arXiv, accessed on May 18, 2025, https://arxiv.org/html/2503.04723
9. Analyzing 16,193 LLM Papers for Fun and Profits - arXiv, accessed on May 18, 2025, https://arxiv.org/html/2504.08619v3
10. Full article: From natural language to simulations: applying AI to automate simulation modelling of logistics systems - Taylor & Francis Online, accessed on May 18, 2025, https://www.tandfonline.com/doi/full/10.1080/00207543.2023.2276811
11. [2408.12159] Search-Based LLMs for Code Optimization - arXiv, accessed on May 18, 2025, https://arxiv.org/abs/2408.12159
12. Search-Based LLMs for Code Optimization - arXiv, accessed on May 18, 2025, https://arxiv.org/html/2408.12159v1
13. Meta AI Introduces Meta LLM Compiler: A State-of-the-Art LLM that Builds upon Code Llama with Improved Performance for Code Optimization and Compiler Reasoning - MarkTechPost, accessed on May 18, 2025, https://www.marktechpost.com/2024/06/28/meta-ai-introduces-meta-llm-compiler-a-state-of-the-art-llm-that-builds-upon-code-llama-with-improved-performance-for-code-optimization-and-compiler-reasoning/
14. Optimizing LLM Inference with Hardware-Software Co-Design - AiThority, accessed on May 18, 2025, https://aithority.com/machine-learning/optimizing-llm-inference-with-hardware-software-co-design/
15. An Inquiry into Datacenter TCO for LLM Inference with FP8 - arXiv, accessed on May 18, 2025, https://arxiv.org/html/2502.01070v3
16. LLM Optimization and Deployment on SiFive RISC-V Intelligence Products, accessed on May 18, 2025, https://www.sifive.com/blog/llm-optimization-and-deployment-on-sifive-intellig
17. How to optimize a mixed stack/register bytecode with control flow and side effects?, accessed on May 18, 2025, https://softwareengineering.stackexchange.com/questions/384347/how-to-optimize-a-mixed-stack-register-bytecode-with-control-flow-and-side-effec
18. General | Programming Optimization - Codecademy, accessed on May 18, 2025, https://www.codecademy.com/resources/docs/general/programming-optimization
19. Meta Large Language Model Compiler: Foundation Models of Compiler Optimization | Research - AI at Meta, accessed on May 18, 2025, https://ai.meta.com/research/publications/meta-large-language-model-compiler-foundation-models-of-compiler-optimization/
20. LLM Compiler: A Game Changer for Software Development - Emergent Behavior,

accessed on May 18, 2025, https://www.emergentbehavior.co/p/2024-06-28-llm-as-compiler
21. As this LLM operates on LLVM intermediate representation language, the result ca... | Hacker News, accessed on May 18, 2025, https://news.ycombinator.com/item?id=40824705
22. Natural Language Outlines for Code: Literate Programming in the LLM Era - arXiv, accessed on May 18, 2025, https://arxiv.org/html/2408.04820v4
23. [2407.05411] Assessing Code Generation with Intermediate Languages - arXiv, accessed on May 18, 2025, https://arxiv.org/abs/2407.05411
24. Virtual Machinations: Leveraging the Linguistic Bytecode of Large Language Models | Erik Meijer - YouTube, accessed on May 18, 2025, https://www.youtube.com/watch?v=ySKS719R6fc
25. Natural language boosts LLM performance in coding, planning, and robotics | MIT News, accessed on May 18, 2025, https://news.mit.edu/2024/natural-language-boosts-llm-performance-coding-planning-robotics-0501
26. \tool: Differential testing with LLMs using Natural Language Specifications and Code Artifacts - arXiv, accessed on May 18, 2025, https://arxiv.org/html/2410.04249v3
27. From 1+1 in Assembly to LLMs: The Evolution of Computing Abstraction | Taewoon Kim, accessed on May 18, 2025, https://taewoon.kim/2024-11-12-1+1/
28. Scaffolded LLMs as natural language computers | Hacker News, accessed on May 18, 2025, https://news.ycombinator.com/item?id=35544964
29. LLM for parsing Natural Language : r/Compilers - Reddit, accessed on May 18, 2025, https://www.reddit.com/r/Compilers/comments/13vhzy5/llm_for_parsing_natural_language/
30. Hot take: AI will probably write code that looks like gibberish to humans (and why that makes sense) - Reddit, accessed on May 18, 2025, https://www.reddit.com/r/ArtificialInteligence/comments/1htlgly/hot_take_ai_will_probably_write_code_that_looks/
31. Can't AI just skip Python and write bytecode? - YouTube, accessed on May 18, 2025, https://www.youtube.com/watch?v=Z2lQ0ieWilY
32. Show HN: Marsha – An LLM-Based Programming Language | Hacker News, accessed on May 18, 2025, https://news.ycombinator.com/item?id=36864021
33. Bytecode - Wikipedia, accessed on May 18, 2025, https://en.wikipedia.org/wiki/Bytecode
34. Understanding Bytecode | Startup House, accessed on May 18, 2025, https://startup-house.com/glossary/bytecode
35. Difference between Byte Code and Machine Code | GeeksforGeeks, accessed on May 18, 2025, https://www.geeksforgeeks.org/difference-between-byte-code-and-machine-code/
36. Byte Code in Java | GeeksforGeeks, accessed on May 18, 2025, https://www.geeksforgeeks.org/byte-code-in-java/

37. Definition of bytecode | PCMag, accessed on May 18, 2025, https://www.pcmag.com/encyclopedia/term/bytecode
38. What is Bytecode? - Codecademy, accessed on May 18, 2025, https://www.codecademy.com/resources/blog/what-is-bytecode/
39. www.techtarget.com, accessed on May 18, 2025, https://www.techtarget.com/whatis/definition/bytecode#:~:text=What%20is%20the%20advantage%20of,interpret%20the%20same%20bytecode%20files.
40. What is Byte Code in Java? Benefits and Drawbacks - The Knowledge Academy, accessed on May 18, 2025, https://www.theknowledgeacademy.com/blog/what-is-byte-code-in-java/
41. Bytecode - EL Passion, accessed on May 18, 2025, https://www.elpassion.com/glossary/bytecode
42. Bytecode vs Machine Code: Understanding the Differences - Shiksha Online, accessed on May 18, 2025, https://www.shiksha.com/online-courses/articles/bytecode-vs-machine-code-understanding-the-differences/
43. What are advantages of bytecode over native code? [closed] - Stack Overflow, accessed on May 18, 2025, https://stackoverflow.com/questions/48144/what-are-advantages-of-bytecode-over-native-code
44. What is the benefit of compiling to bytecode before interpreting in interpreted languages like Java? : r/AskProgramming - Reddit, accessed on May 18, 2025, https://www.reddit.com/r/AskProgramming/comments/cwk8xu/what_is_the_benefit_of_compiling_to_bytecode/
45. Can you explain the difference between bytecode and machine code? Are they simply different levels of abstraction for computing instructions to be executed on hardware? - Quora, accessed on May 18, 2025, https://www.quora.com/Can-you-explain-the-difference-between-bytecode-and-machine-code-Are-they-simply-different-levels-of-abstraction-for-computing-instructions-to-be-executed-on-hardware
46. Introduction to Computer Programs: Bytecode and Machine Code - Think Object Oriented in Java and C#, accessed on May 18, 2025, https://thinkobjectoriented.hashnode.dev/introduction-to-computer-programs-bytecode-machine-code
47. Bytecode vs. Machine Code | Pure Storage Blog, accessed on May 18, 2025, https://blog.purestorage.com/purely-educational/bytecode-vs-machine-code/
48. Understanding Java Compilation: From Bytecodes to Machine Code in the JVM - Azul, accessed on May 18, 2025, https://www.azul.com/blog/understanding-java-compilation-from-bytecodes-to-machine-code/
49. Understanding Bytecode and Java Virtual Machines - CodeProject, accessed on May 18, 2025, https://www.codeproject.com/Articles/5272848/Understanding-Bytecode-and-Java-Virtual-Machines
50. Java Virtual Machine (JVM): Introduction & Its Architecture - Turing, accessed on

May 18, 2025, https://www.turing.com/kb/java-virtual-machine
51. What Happens After We Hit Compile in Java? - belief driven design, accessed on May 18, 2025, https://belief-driven-design.com/what-happens-after-compile-java-315fe/
52. Intermediate Representation - ACM Queue, accessed on May 18, 2025, https://queue.acm.org/detail.cfm?id=2544374
53. Accessible and Portable LLM Inference by Compiling Computational Graphs into SQL - arXiv, accessed on May 18, 2025, https://arxiv.org/pdf/2502.02818
54. DiffSpec: Differential testing with LLMs using Natural Language Specifications and Code Artifacts - arXiv, accessed on May 18, 2025, https://arxiv.org/pdf/2410.04249
55. MLC-LLM: Universal LLM Deployment Engine with ML Compilation, accessed on May 18, 2025, https://blog.mlc.ai/2024/06/07/universal-LLM-deployment-engine-with-ML-compilation
56. Creating and building out a cross-platform app via LLM in <15 minutes without writing a line of code - Reddit, accessed on May 18, 2025, https://www.reddit.com/r/ChatGPTCoding/comments/1eexu7q/creating_and_building_out_a_crossplatform_app_via/
57. The Rise and Fall of ONNX (feat. PyTorch 2.0) - SqueezeBits, accessed on May 18, 2025, https://blog.squeezebits.com/the-rise-and-fall-of-onnx-feat-pytorch-20-42184
58. Meta AI Introduces Meta LLM Compiler: A State-of-the-Art LLM that Builds upon Code Llama with Improved Performance for Code Optimization and Compiler Reasoning : r/machinelearningnews - Reddit, accessed on May 18, 2025, https://www.reddit.com/r/machinelearningnews/comments/1dqn8to/meta_ai_introduces_meta_llm_compiler_a/
59. Java Bytecode Verification with OCL Why, How and When? - The Journal of Object Technology, accessed on May 18, 2025, https://www.jot.fm/issues/issue_2020_03/article13.pdf
60. Bytecode design resources? : r/ProgrammingLanguages - Reddit, accessed on May 18, 2025, https://www.reddit.com/r/ProgrammingLanguages/comments/fievyz/bytecode_design_resources/
61. Universal high-level intermediate representation : r/ProgrammingLanguages - Reddit, accessed on May 18, 2025, https://www.reddit.com/r/ProgrammingLanguages/comments/oz52lu/universal_highlevel_intermediate_representation/
62. What are the dangers of bytecode manipulation (if any)? - Stack Overflow, accessed on May 18, 2025, https://stackoverflow.com/questions/4193689/what-are-the-dangers-of-bytecode-manipulation-if-any
63. Show HN: Letting LLMs Run a Debugger | Hacker News, accessed on May 18, 2025, https://news.ycombinator.com/item?id=43023698
64. Which compiled language do LLMs understand well? Are there some that understand bytecode or binary? : r/ChatGPTCoding - Reddit, accessed on May

18, 2025, https://www.reddit.com/r/ChatGPTCoding/comments/1gw13nb/which_compiled_language_do_llms_understand_well/
65. 5 Advanced Java Debugging Techniques Every Developer Should Know About - InfoQ, accessed on May 18, 2025, https://www.infoq.com/articles/Advanced-Java-Debugging-Techniques/
66. iSEngLab/AwesomeLLM4SE: A Survey on Large Language Models for Software Engineering - GitHub, accessed on May 18, 2025, https://github.com/iSEngLab/AwesomeLLM4SE
67. Bytecode Corruption Attacks Are Real — And How To Defend Against Them, accessed on May 18, 2025, https://people.cs.kuleuven.be/~stijn.volckaert/papers/2018_DIMVA_Interpreters.pdf
68. (PDF) How secure is AI-generated code: a large-scale comparison of large language models - ResearchGate, accessed on May 18, 2025, https://www.researchgate.net/publication/387306336_How_secure_is_AI-generated_code_a_large-scale_comparison_of_large_language_models
69. The Hidden Risks of LLM-Generated Web Application Code: A Security-Centric Evaluation of Code Generation Capabilities in Large Language Models - arXiv, accessed on May 18, 2025, https://arxiv.org/html/2504.20612v1
70. The Hidden Risks of LLM-Generated Web Application Code : r/PromptEngineering - Reddit, accessed on May 18, 2025, https://www.reddit.com/r/PromptEngineering/comments/1kb5xmj/the_hidden_risks_of_llmgenerated_web_application/
71. Adversarial Misuse of Generative AI | Google Cloud Blog, accessed on May 18, 2025, https://cloud.google.com/blog/topics/threat-intelligence/adversarial-misuse-generative-ai
72. Security of LLMs and LLM systems: Key risks and safeguards - Red Hat, accessed on May 18, 2025, https://www.redhat.com/en/blog/llm-and-llm-system-risks-and-safeguards
73. Anatomy of an LLM RCE - CyberArk, accessed on May 18, 2025, https://www.cyberark.com/resources/threat-research-blog/anatomy-of-an-llm-rce
74. Sallm: Security Assessment of Generated Code - arXiv, accessed on May 18, 2025, https://arxiv.org/pdf/2311.00889
75. Why eBPF is Secure: A Look at the Future Technology in LLM Security - Protect AI, accessed on May 18, 2025, https://protectai.com/blog/why-ebpf-is-secure
76. [2503.15554] A Comprehensive Study of LLM Secure Code Generation - arXiv, accessed on May 18, 2025, https://arxiv.org/abs/2503.15554
77. LLM4Decompile: Decompiling Binary Code with LLM - Hacker News, accessed on May 18, 2025, https://news.ycombinator.com/item?id=39733275
78. Your favorite programming language will be dead soon... : r/singularity - Reddit, accessed on May 18, 2025, https://www.reddit.com/r/singularity/comments/1jud2tk/your_favorite_programmi

ng_language_will_be_dead/
79. Do LLM work better with syntax-heavy languages? : r/ChatGPTCoding - Reddit, accessed on May 18, 2025, https://www.reddit.com/r/ChatGPTCoding/comments/1hex7te/do_llm_work_better_with_syntaxheavy_languages/
80. Translating 10M lines of Java to Kotlin - Hacker News, accessed on May 18, 2025, https://news.ycombinator.com/item?id=42452640
81. Best AI Programming Languages: Python, R, Julia & More - SitePoint, accessed on May 18, 2025, https://www.sitepoint.com/best-programming-language-for-ai/
82. Top AI Programming Languages for Advanced Development in 2025 - eLuminous Technologies, accessed on May 18, 2025, https://eluminoustechnologies.com/blog/ai-programming-languages/
83. 6 Best programming languages for AI development - Academy SMART, accessed on May 18, 2025, https://academysmart.com/insights/6-best-programming-languages-for-ai-development/
84. AI and deep learning define the future of programming, will Kotlin fly or die?, accessed on May 18, 2025, https://discuss.kotlinlang.org/t/ai-and-deep-learning-define-the-future-of-programming-will-kotlin-fly-or-die/2264
85. Intermediate Code Generation in Compiler Design | GeeksforGeeks, accessed on May 18, 2025, https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/
86. LLMs struggle to write performant code - CodeFlash AI, accessed on May 18, 2025, https://www.codeflash.ai/post/llms-struggle-to-write-performant-code
87. How efficient is LLM-generated code? A rigorous & high-standard benchmark - OpenReview, accessed on May 18, 2025, https://openreview.net/forum?id=suz4utPr9Y
88. How Efficient is LLM-Generated Code? A Rigorous & High-Standard Benchmark - arXiv, accessed on May 18, 2025, https://arxiv.org/abs/2406.06647
89. Evaluating the Performance of LLM-Generated Code for ChatGPT-4 and AutoGen Along with Top-Rated Human Solutions, accessed on May 18, 2025, https://www.dre.vanderbilt.edu/~schmidt/PDF/ChatGPT_vs__Stack_Overflow_Performance.pdf
90. Assessing Code Generation with Intermediate Languages - arXiv, accessed on May 18, 2025, http://arxiv.org/pdf/2407.05411
91. CodeBridge: The Universal Code Translation Platform - DEV Community, accessed on May 18, 2025, https://dev.to/aniruddhaadak/codebridge-the-universal-code-translation-platform-1bid
92. Assessing Code Generation with Intermediate Languages - arXiv, accessed on May 18, 2025, https://arxiv.org/pdf/2407.05411
93. Modifying an existing code LLM for bytecode ? : r/MLQuestions - Reddit, accessed on May 18, 2025,

https://www.reddit.com/r/MLQuestions/comments/10gezlw/modifying_an_existing_code_llm_for_bytecode/
94. ByteCodeLLM – Privacy in the LLM Era: Byte Code to Source Code - CyberArk, accessed on May 18, 2025, https://www.cyberark.com/resources/threat-research-blog/bytecodellm-privacy-in-the-llm-era-byte-code-to-source-code
95. Show HN: I built a hardware processor that runs Python | Hacker News, accessed on May 18, 2025, https://news.ycombinator.com/item?id=43820228
96. cyberark/ByteCodeLLM - GitHub, accessed on May 18, 2025, https://github.com/cyberark/ByteCodeLLM
97. Can Large Language Models Understand Intermediate Representations? - arXiv, accessed on May 18, 2025, https://arxiv.org/html/2502.06854v1
98. Enhancing the software ecosystem with LLMs | Computing - Lawrence Livermore National Laboratory, accessed on May 18, 2025, https://computing.llnl.gov/about/newsroom/enhancing-software-ecosystem-llms
99. Introduction to Large Language Models | Machine Learning - Google for Developers, accessed on May 18, 2025, https://developers.google.com/machine-learning/resources/intro-llms
100. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future - arXiv, accessed on May 18, 2025, https://arxiv.org/html/2408.02479v1/
101. Application of Large Language Models (LLMs) in Software Engineering: Overblown Hype or Disruptive Change? - SEI Blog, accessed on May 18, 2025, https://insights.sei.cmu.edu/blog/application-of-large-language-models-llms-in-software-engineering-overblown-hype-or-disruptive-change/
102. Patchscopes: A unifying framework for inspecting hidden representations of language models - Google Research, accessed on May 18, 2025, https://research.google/blog/patchscopes-a-unifying-framework-for-inspecting-hidden-representations-of-language-models/
103. The Transformative Influence of Large Language Models on Software Development, accessed on May 18, 2025, https://www.researchgate.net/publication/376077151_The_Transformative_Influence_of_Large_Language_Models_on_Software_Development