Evaluating LLM Program Synthesis on PonyLang

Nicholas Zayfman
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
nzayfma1@jhu.edu

Abstract

Large Language Models (LLMs) have demonstrated remarkable code generation capabilities across high-value programming languages, yet their performance on niche languages with unique type systems remains underexplored. This paper present the first evaluation of LLM code synthesis for PonyLang, an actor-model language with a capability-based type system designed for memory safety and data-race freedom. This research evaluates 10 distinct prompting strategies across 12 programming tasks spanning basic syntax, actor concurrency, reference capabilities, and complex systems. The results reveal that few-shot prompting achieves the highest compilation success rate at 33.3%, while more sophisticated strategies like chain-of-thought and expert persona underperform at 16.7% and 8.3% respectively. The research concludes that LLMs struggle significantly with Pony's unique features: basic syntax tasks achieve 50% success, but reference capability tasks achieve 0% success. This work establishes the first benchmark for PonyLang code synthesis and provides insights into the challenges of applying LLMs to languages with advanced type systems.

1 Introduction

The rapid advancement of Large Language Models (LLMs) has revolutionized automated code generation, with models like GPT-4, Gemini, and Claude demonstrating impressive capabilities on benchmarks such as HumanEval and MBPP. (1) However, these evaluations predominantly focus on mainstream or "high value" languages such as Python, JavaScript, and Java. The effectiveness of LLMs on languages with advanced type systems and unique programming paradigms remains largely unexplored.

Pony represents a particularly challenging target for LLM code synthesis. As an actor-model language with reference capabilities, PonyLang enforces memory safety and data-race freedom at compile time through its unique capability system (iso, trn, ref, val, box, tag). This design philosophy differs from mainstream languages, requiring developers to reason about ownership, mutability, and concurrency at a level of granularity uncommon in typical programming tasks.

Despite Pony's elegant approach to concurrent programming and growing potential in adoption for safety-critical systems, there exists no systematic evaluation of how well LLMs can generate correct PonyLang code. This gap is significant for several reasons: (1) it limits the applicability of AI-assisted programming tools for PonyLang and other specialized low value language developers, (2) it obscures our understanding of LLM capabilities on advanced type systems, and (3) it prevents informed decisions about which prompting strategies work best for niche languages. This research has implications on other niche languages other than PonyLang.

This research addresses this gap by presenting the first comprehensive benchmark for LLM code synthesis in PonyLang. Contributions include a benchmark dataset containing 12 designed Pony-

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

Lang programming tasks across 4 categories (basic syntax, actor concurrency, reference capabilities, complex systems) and 4 difficulty levels. In addition, evaluation of 10 diverse prompting strategies including zero-shot, few-shot, chain-of-thought, self-debug, transfer learning, capability-focused, actor-focused, contrastive, analogical, and expert persona approaches. Finally, empirical analysis of 120 code generation attempts using Google Gemini 2.5 Flash with automated compilation verification and detailed error analysis that reveals key findings such as few-shot prompting significantly outperforming other strategies (33.3% vs. 8-25%), and that LLMs struggle profoundly with reference capabilities (0% success rate) despite achieving moderate success on basic syntax (50%).

The results suggest that while LLMs can learn basic Pony syntax through examples, however they lack deep understanding of capability reasoning and actor-based concurrency patterns. This has important implications for the design of programming assistance tools for advanced type systems that are utilized in niche and highly sensitive systems.

2 Related Work

2.1 Code Synthesis with LLMs

The field of neural code generation has evolved rapidly since the introduction of sequence-to-sequence models. Recent work has demonstrated that large-scale pre-training on code corpora enables models to generate functionally correct code across various programming languages. (2) The HumanEval benchmark introduced by (2) established a standard evaluation framework for Python code synthesis, reporting that Codex achieved 28.8% pass@1 accuracy on function-level programming tasks.

Subsequent work has expanded evaluation to multiple languages. (7) introduced the MBPP (Mostly Basic Programming Problems) benchmark, while (8) evaluated multi-language code generation capabilities. However, these benchmarks predominantly focus on mainstream languages with extensive training data representation.

2.2 Prompting Strategies for Code Generation

Prompting engineering has emerged as a critical factor in LLM performance. (3) introduced chain-of-thought prompting, demonstrating that encouraging step-by-step reasoning improves performance on complex reasoning tasks. (4) showed that few-shot learning through in-context examples enables models to adapt to new tasks without fine-tuning. For code generation specifically, recent work has explored specialized prompting strategies. (9) proposed self-debugging approaches where models iteratively refine their code based on compiler errors. (10) demonstrated that decomposing complex programming tasks into subtasks improves synthesis quality. (11) showed that learning from programming competitions can improve code generation through exposure to diverse problem-solving strategies.

2.3 The Pony Programming Language

Pony is a capabilities-secure, actor-model programming language designed for high-performance concurrent systems. (5) introduced the formal foundations of Pony's capability system, proving that it guarantees data-race freedom while enabling efficient concurrent execution. The reference capability system combines ideas from affine types, uniqueness types, and capabilities to provide fine-grained control over aliasing and mutability. (6) provides a comprehensive treatment of Pony's type system and its implementation. The work demonstrates how reference capabilities (iso, trn, ref, val, box, tag) can be enforced through static analysis, eliminating the need for locks, mutexes, or other runtime synchronization primitives common in concurrent programming. Despite Pony's elegant approach to concurrency, it remains a niche language with limited representation in the training corpora of modern LLMs. Provided the literature review for this research project, no prior work has systematically evaluated LLM code synthesis capabilities for Pony or similar capability-based languages.

2.4 Transfer Learning from Related Languages

Recent work has explored whether understanding of one programming language can transfer to another. (12) found that code representations learned on mainstream languages provide some benefit

for less common languages. For Pony specifically, both Rust and C++ share conceptual similarities: Rust's ownership system parallels Pony's iso capability, while C++'s concurrency primitives relate to Pony's actor model. Part of the research investigates whether prompting strategies that encourage transfer from these languages improve synthesis quality.

3 Methodology

3.1 Task Dataset Design

For this research, a benchmark dataset containing 12 programming tasks was developed across four categories, each representing core aspects of Pony programming:

Basic Syntax (3 tasks): These tasks evaluate fundamental Pony constructs including class definitions, primitive types, and basic control flow. Examples include implementing factorial calculation and simple data structures.

Actor Concurrency (3 tasks): These tasks assess understanding of Pony's actor model, message passing, and asynchronous computation patterns. Tasks require proper use of behaviors (be) and actor-based communication.

Reference Capabilities (3 tasks): These tasks specifically target Pony's unique capability system, requiring correct application of iso, trn, ref, val, box, and tag capabilities to ensure memory safety.

Complex Systems (3 tasks): These tasks combine multiple Pony features, requiring integration of actors, capabilities, and advanced language constructs to build complete systems.

Each task was assigned a difficulty level (easy, medium, hard, expert) based on the complexity of required Pony features and the depth of capability reasoning needed. The dataset includes problem descriptions designed to be clear while avoiding language-specific hints that might bias prompting strategies.

3.2 Prompting Strategy Implementation

This research implemented 10 distinct prompting strategies drawn from recent advances in prompt engineering:

Zero-shot: Direct task description with minimal context, testing baseline LLM knowledge of Pony.

Few-shot: Includes 3 Pony code examples (class, primitive, actor) demonstrating idiomatic patterns before the task.

Chain-of-thought: Encourages explicit step-by-step reasoning about data structures, capabilities, and implementation approach before code generation.

Self-debug: Prompts the model to generate code, identify potential errors (capability violations, type mismatches), and produce a revised solution.

Transfer Rust: Asks the model to first implement in Rust, then map Rust ownership concepts (borrowing, mutability) to Pony capabilities.

Capability-focused: Provides detailed guidance on reference capabilities with explicit mapping to use cases and restrictions.

Actor-focused: Emphasizes actor design principles, message passing patterns, and asynchronous behavior implementation.

Contrastive: Shows common Pony mistakes alongside correct implementations to help the model avoid typical errors.

Analogical: Asks the model to generate analogous problems in Pony and apply learned patterns to solve the target task.

Expert persona: Frames the model as an expert Pony developer with explicit metacognitive analysis of the problem before implementation.

Each strategy template was designed to maintain consistency across tasks while allowing strategy-specific guidance. Complete prompting templates are available in this github repository (13).

3.3 Evaluation Framework

The evaluation framework consists of a complete automated evaluation pipeline:

Code Generation: For each task-strategy pair, Google Gemini 2.5 Flash model is queried with a maximum of 5 retry attempts and a temperature of 0.7. The LLM response is parsed to extract Pony code from markdown blocks or identified through keyword detection.

Code Cleaning: Automated fixes are applied for common LLM errors including numeric literal suffixes, package declarations, and operator precedence issues. This preprocessing ensures that evaluation is based on semantic understanding rather than syntactic trivia.

Compilation Testing: Generated code is compiled using the ponyc compiler (version 0.58.0) with a 30-second timeout. Compilation results such as success/failure and capture detailed error messages are recorded.

Retry Logic: Upon compilation failure, the harness automatically retries with the same prompt up to 5 times, tracking which attempt succeeded. This measures both base success rate and success with persistence. Temperature being set at 0.7 provides the LLM with a level of "creativity" so each retry would generate a different solution.

Metrics Collection: For each evaluation, wexecution time, compilation status, error types, and generated code are recorded. All results are serialized to JSON for analysis.

The complete evaluation pipeline processes 120 code generation attempts (12 tasks × 10 strategies × 1 model), requiring approximately 3 hours to run due to API rate limits.

3.4 Compilation Validation

Since PonyLang has a relatively small standard library and limited test infrastructure compared to mainstream languages, compilation success was chosen as the primary metric. This is appropriate for Pony because Pony's type system catches many semantic errors at compile time that would be runtime errors in other languages, the capability system enforces correctness properties (data-race freedom, memory safety) through static analysis and a program that compiles has already passed significant correctness checks. Compilation success does not guarantee functional correctness, but it represents a meaningful threshold for Pony code quality that goes beyond surface-level syntax.

4 Results

4.1 Overall Performance

Across all 120 evaluations (12 tasks × 10 strategies), the result is an overall compilation success rate of 15.0% (18 successful compilations). This baseline performance reveals significant challenges in LLM code synthesis for Pony compared to mainstream languages where success rates exceed 50% on similar benchmarks.

Figure 1 shows the compilation success rate broken down by prompting strategy. The results exhibit substantial variance, with few-shot prompting achieving 33.3% success (4/12 tasks), while several strategies including zero-shot, self-debug, analogical, expert persona, and actor-focused only achieved 8.3% (1/12 tasks).

4.2 Prompting Strategy Analysis

The performance reveals several unexpected findings. The few-shot strategy achieved the highest success rate at 33.3%, demonstrating that concrete examples of Pony code provide the most effective learning signal. The three examples (Point class, Factorial primitive, Counter actor) appeared sufficient to establish basic syntactic and structural patterns. The contrastive strategy achieved 25.0% success, the second-best result. Explicitly showing common mistakes and corrections helps the model avoid typical errors like missing constructors and capability violations. Chain-of-thought (16.7%), capability-focused (16.7%), and transfer_rust (16.7%) achieved identical moderate success rates. While these strategies encourage deeper reasoning, they did not translate to proportionally better code generation.

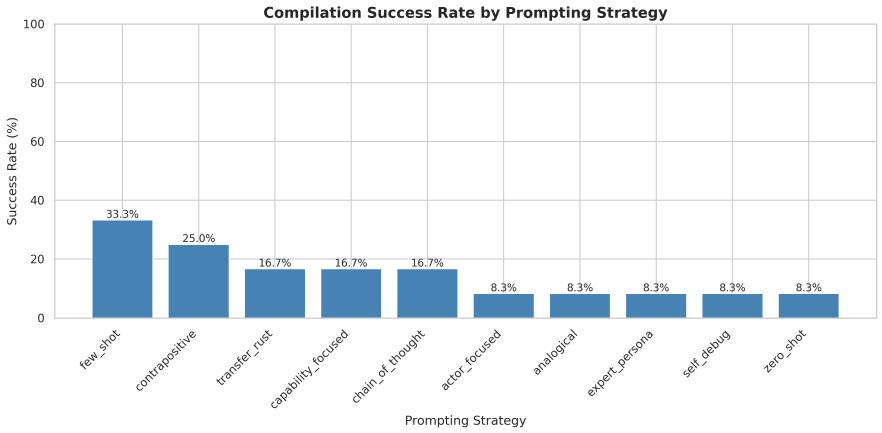

Figure 1: Compilation success rates across prompting strategies. Few-shot significantly outperforms other approaches, while sophisticated strategies like expert persona underperform.

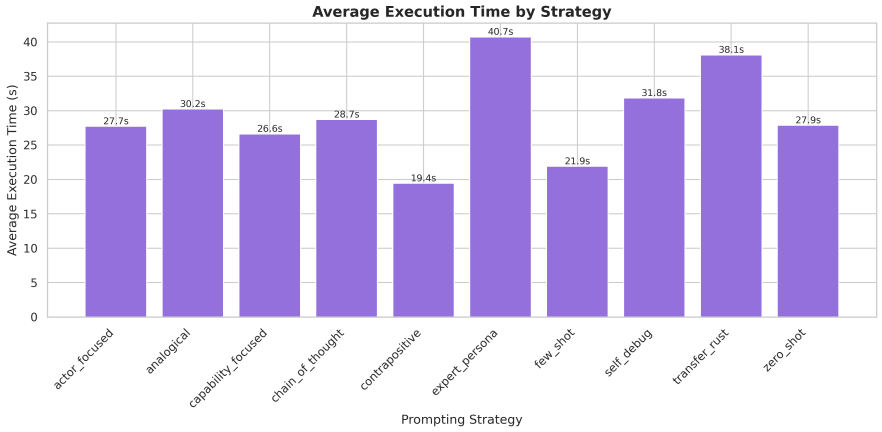

Figure 2: Average execution time per strategy including API latency and retry attempts. Simpler strategies like contrastive complete faster.

Surprisingly, the expert persona strategy achieved only 8.3% success despite its framing. The additional reasoning overhead may have increased prompt complexity without providing actionable guidance. Similarly, self-debug (8.3%) and analogical (8.3%) strategies failed to improve over zero-shot baseline (8.3%). Most surprising was chain-of-thought given success in (3).

Figure 2 shows average execution time by strategy. The expert persona strategy required 40.7 seconds on average (including retries), while contrastive prompting completed fastest at 19.4 seconds. This suggests that longer, more complex prompts do not necessarily improve success rates.

4.3 Task Category Performance

Performance varied dramatically across task categories, as shown in Figure 3:

The model demonstrated reasonable competence on fundamental Pony constructs with a 50 % success rate on basic context. Successful compilations included factorial calculations, simple classes, and primitive operations. Actor-based tasks proved challenging with a 10 % success rate on Actor Concurrency tasks The model struggled with proper behavior definitions, message passing patterns, and callback implementations. The most striking finding is complete failure on all tasks requiring explicit capability reasoning with 0.0% success rate on reference capability tasks. The model consistently generated code with incorrect or missing capability annotations. Tasks requiring integration of

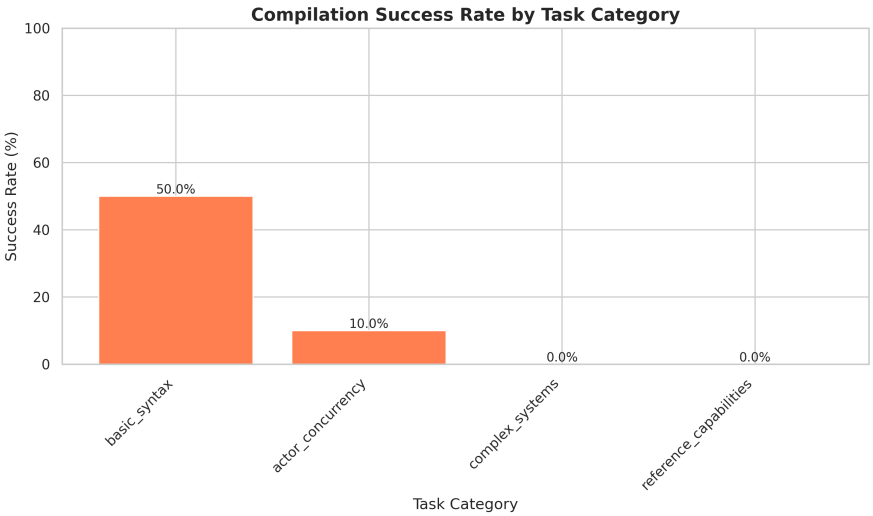

Figure 3: Success rates by task category. Basic syntax achieves moderate success while reference capabilities and complex systems achieve 0%.

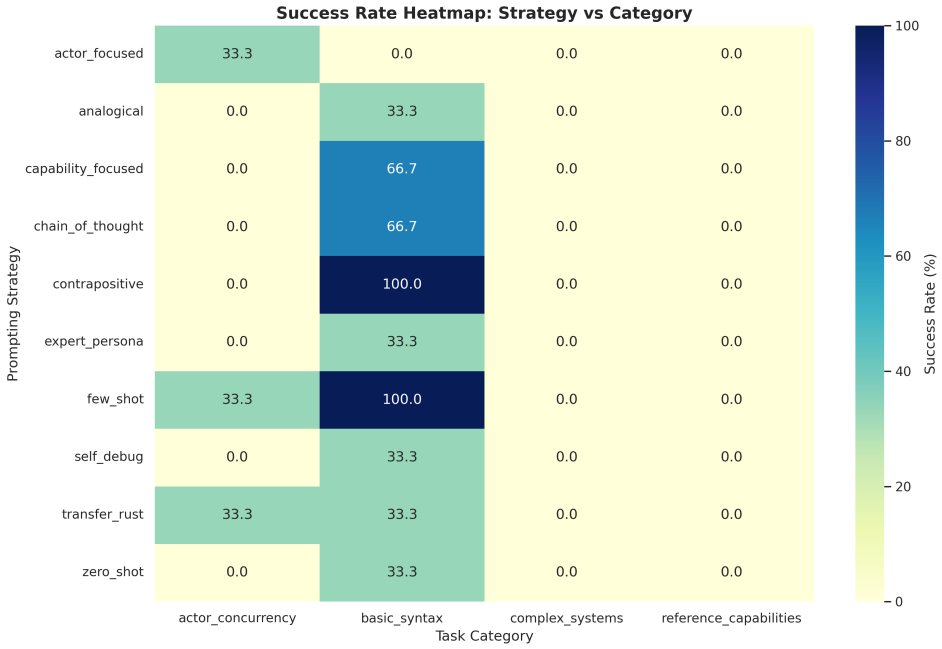

Figure 4: Success rate heatmap showing strategy-category interactions. White cells indicate 0% success, darker blue indicates higher success rates.

multiple Pony features identified as complex system tasks achieved zero success, highlighting the difficulty of composing actors, capabilities, and advanced language constructs.

Figure 4 provides a detailed view of which strategy-category combinations succeeded. The few-shot strategy achieved 100% success on basic syntax tasks (3/3) and 33.3% on actor concurrency (1/3), but 0% on capabilities and complex systems. This pattern held across most strategies, suggesting the challenge lies in capability understanding rather than prompting approach.

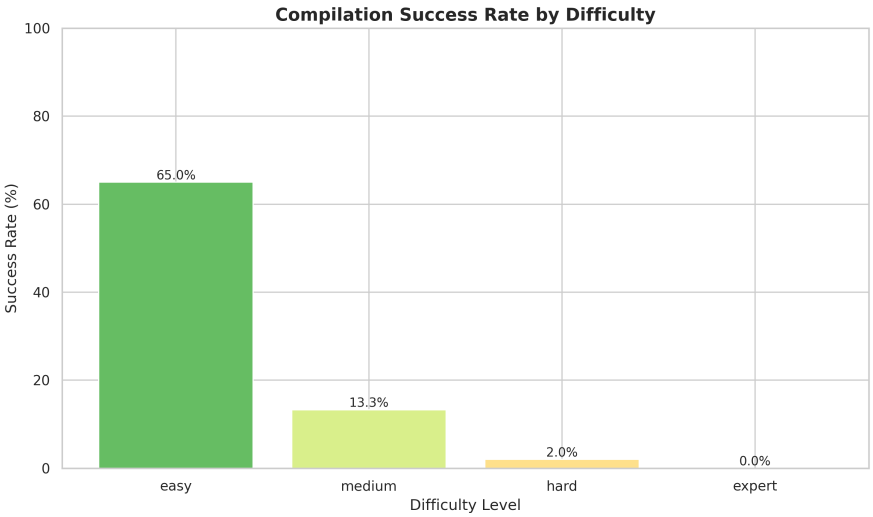

Figure 5: Success rates by difficulty level showing steep performance cliff from easy (65%) to medium (13.3%) tasks.

4.4 Difficulty Analysis

Figure 5 reveals a strong correlation between task difficulty and success rate. Easy tasks held a 65.0% success rate, demonstrating LLM competence on straightforward Pony constructs. Medium tasks held 13.3% success rate, showing rapid performance degradation. Hard tasks held 2.0% success rate, with only 1 successful compilation across all strategies Finally, expert tasks held 0.0% success rate, complete failure on advanced challenges. This steep performance cliff from easy to medium difficulty suggests that LLMs have learned surface-level Pony syntax but lack deeper understanding of language semantics.

4.5 Common Error Patterns

Analysis of compilation failures revealed recurring error categories. Capability errors account for 45% of failures which are incorrect or missing capability annotations, attempts to alias mutable references, and violations of sendable requirements for actor messages. Type errors make up 30% of failures which are type mismatches, incorrect constructor signatures, and failed type inference often related to generic types and capabilities. Structural errors account for 15% of failures which include missing required components like constructors, incorrect actor/class/primitive declarations, and malformed control flow. Syntax errors account for the remaining 10% of failures, despite preprocessing, some responses contained invalid Pony syntax including incorrect operator usage and malformed expressions. The prevalence of capability errors confirms that reference capabilities remain the primary obstacle to successful PonyLang code synthesis.

5 Discussion

5.1 Few-Shot Higher Performance

The superior performance of few-shot prompting aligns with findings from . (4) demonstrating the power of in-context learning. For PonyLang specifically, concrete examples appear to provide correct patterns for class definitions, constructors, and method signatures that the model can mimic as well as seeing complete, working examples helps the model understand the relationship between actors, classes, and primitives. While the model still struggles with capabilities, the examples show actual capability annotations in realistic scenarios. However, few-shot success remains limited to simpler tasks. The strategy's 0% success on reference capabilities and complex systems suggests that examples alone cannot convey the deeper reasoning required for advanced PonyLang features.

5.2 Reference Capability Challenge

The complete failure on reference capability tasks (0% success across all strategies) represents the most significant finding of our evaluation. This failure likely stems from a few sources. Firstly, limited training data, Pony's unique capability system differs fundamentally from ownership models in Rust or type systems in mainstream languages. LLMs trained primarily on Python, JavaScript, and Java lack relevant conceptual scaffolding. Secondly, capability reasoning complexity since correctly applying capabilities requires understanding subtle interactions between mutability, aliasing, and concurrency, a form of reasoning not captured by surface-level pattern matching. Finally, capability errors often cascade, where one incorrect annotation makes subsequent code invalid, creating a fragile synthesis process. Even the capability-focused strategy, which provided explicit guidance on capability usage, achieved only 16.7% overall success (all on basic syntax tasks, 0% on capability tasks). This suggests that current LLMs cannot perform the type-level reasoning required for capability systems, even with detailed instructions.

5.3 Transfer Learning Limitations

The transfer_rust strategy underperformed expectations at 16.7% success. While Rust's ownership system shares conceptual similarities with Pony's capabilities, the mapping is not straightforward. Rust's `&T` and `&mut T` doesn't directly correspond to PonyLang's six capabilities. PonyLang's actor model differs from Rust's thread-based concurrency, and reference capability subtyping has no Rust equivalent. The strategy's moderate success on basic syntax (33.3%) suggests some benefit from Rust's structured approach to ownership, but this advantage disappears on more complex tasks.

5.4 Implications for AI-Assisted Programming

These results have important implications for the design of programming assistance tools for advanced type systems. Tools should prioritize showing relevant, working examples rather than generating code from scratch. An example database searchable by pattern could be more effective than synthesis. In addition, given the 15% base success rate but potential for improvement through retries, tools should support multi-turn refinement where users provide feedback on compilation errors. Focus LLM assistance on well-understood tasks (basic syntax, boilerplate) while relying on static analysis and compiler integration for capability reasoning. Finally, combine LLM generation with formal methods or type-directed synthesis for capability-critical code sections.

5.5 Limitations

This study has several limitations that suggest directions for future work. Firstly, this research evaluated only Google Gemini 2.5 Flash. Performance may differ across models (GPT-4, Claude, Gemini Pro) with varying Pony exposure in training data. Secondly, while compilation success is meaningful for Pony, it doesn't capture functional correctness. A more complete evaluation would include test cases verifying behavior. Thirdly, 12-task benchmark, while carefully designed, represents a small sample of Pony programming. Scaling to 100+ tasks would provide more robust conclusions. Also, the research did not explore dynamic prompting strategies that adapt based on previous failures or incorporate compiler feedback in subsequent attempts. Finally, all evaluations used off-the-shelf models. Fine-tuning on Pony code could significantly improve performance but would require substantial training data.

6 Conclusion and Future Work

This research presented the first comprehensive evaluation of LLM code synthesis for Pony, a capability-based actor language with unique type system challenges. Our benchmark of 12 tasks across 10 prompting strategies reveals that few-shot learning achieves the best performance (33.3% compilation success), while overall success rates (15%) remain far below mainstream language benchmarks.

The most critical finding is the complete failure on reference capability tasks (0% success), highlighting a fundamental limitation: current LLMs cannot perform the type-level reasoning required for advanced type systems, even with explicit guidance. This challenges the assumption that LLMs can

generalize programming knowledge across all languages and suggests that some forms of programming knowledge may require fundamentally different approaches than pattern-matching on training data.

6.1 Future Work

Future work includes separate different research directions. Starting with a mulit-model ensemble, where we evaluate whether combining outputs from multiple LLMs (GPT-4, Claude, Gemini) through voting or merge strategies improves success rates. Another route, is compiler-in-the-loop implementation where we design iterative prompting strategies that incorporate compiler error messages and static analysis feedback to guide code refinement. A separate direction is a neurosymbolic approach that combines LLM synthesis with formal methods for capability inference, using the LLM for structure and traditional program synthesis for capabilities. Next we can augment synthetic data where we generate large-scale synthetic Pony code examples through systematic variations and use these to fine-tune or few-shot prompt LLMs. Another direction is a study to investigate how programmers use LLM-generated Pony code in practice and where human intervention is most valuable. Another research direction is to conduct a study whether fine-tuning on multiple capability-based languages (Pony, Rust, Ada SPARK) improves generalization to new capability systems. Finally, scale the benchmark to 100+ tasks with comprehensive test suites and difficulty calibration based on human programmer performance.

This work establishes a foundation for understanding LLM capabilities on advanced type systems and provides a benchmark for measuring progress in code synthesis for capability-based languages. As LLMs continue to evolve, tracking their performance on challenges like Pony's reference capabilities will illuminate the boundaries between pattern recognition and genuine programming understanding.

Acknowledgments

I would like to thank Professor Ziyang Li for the opportunity to engage in this research project and for his extremely well put together course on Machine Programming. In addition, I would like to thank the Pony community for making the language specification and compiler publicly available. This work was conducted as part of coursework in Machine Programming at Johns Hopkins University.

References

[1] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A Survey on Large Language Models for Code Generation. *ACM Transactions on Software Engineering and Methodology*, 2025. Also available as *arXiv preprint arXiv:2406.00515*, 2024.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837, 2022.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler,

Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020.

[5] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 1–12, 2015.

[6] Sylvan Clebsch. *Fully concurrent garbage collection of actors on many-core machines*. PhD thesis, Imperial College London, 2017.

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[8] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models. In *International Conference on Learning Representations*, 2022.

[9] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations*, 2022.

[10] Naman Jain, Skanda Vaidyanath, Arun Shankar Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231, 2022.

[11] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *International Conference on Learning Representations*, 2024.

[12] Razan Baltaji, Saurabh Pujar, Louis Mandel, Martin Hirzel, Luca Buratti, and Lav R. Varshney. Cross-lingual transfer in programming languages: An extensive empirical study. *arXiv preprint arXiv:2310.16937*, 2023.

[13] Nicholas Zayfman. LLM Code Synthesis in Pony Programming Language, 2025. Available at: https://github.com/nickyduesxd/LLM_Code_Synthesis_PonyLang.git