From Tool Calling to Symbolic Thinking: LLMs in a Persistent Lisp Metaprogramming Loop*

Jordi de la Torre†
Ph.D. in Computer Science and Mathematics of Security
Barcelona, ES
jordi.delatorre@gmail.com

June 13, 2025

Abstract

We propose a novel architecture for integrating large language models (LLMs) with a persistent, interactive Lisp environment. This setup enables LLMs to define, invoke, and evolve their own tools through programmatic interaction with a live REPL. By embedding Lisp expressions within generation and intercepting them via a middleware layer, the system allows for stateful external memory, reflective programming, and dynamic tool creation. We present a design framework and architectural principles to guide future implementations of interactive AI systems that integrate symbolic programming with neural language generation.

1 Introduction

Large language models have demonstrated remarkable capabilities across a wide range of tasks, especially when augmented with external tools and memory. Most current systems rely on predefined APIs, toolsets, or retrieval pipelines. These static approaches, while powerful, limit the model's ability to construct or evolve its own tool environment.

This work proposes an alternative path: empowering language models to use a Lisp REPL as a persistent programming and reasoning environment. Lisp's symbolic flexibility, rich history in AI, and reflective capabilities make it a natural choice. Our system enables the LLM to define and call Lisp functions dynamically during generation, maintaining state across sessions and supporting structured reasoning beyond pure text generation.

2 Background

Symbolic artificial intelligence (AI) traces its roots to early efforts aimed at formalizing reasoning and knowledge manipulation. John McCarthy's pioneering development of Lisp laid a foundational framework for symbolic computation, enabling advances in common-sense reasoning and recursive function theory [1, 2]. This symbolic tradition was furthered by systems such as Winograd's SHRDLU, which showcased procedural representations for natural language understanding [3],

*This paper presents a conceptual framework intended to guide future implementations rather than report experimental results.

†mailto:jordi.delatorre@gmail.com

and Minsky's conceptualization of frames and semantic networks, offering models for human knowledge structures [4].

Concurrently, the theoretical underpinnings of symbolic metaprogramming were shaped by Barendregt's formalization of the lambda calculus [5] and later expanded by Hankin's rigorous treatments of program analysis and transformation [6]. These foundational concepts fueled the rise of expert systems, many implemented in Lisp, such as Mycin, a rule-based system for medical inference [7], and Common Lisp, which standardized features for scalable symbolic applications [8].

Despite their clear rules and interpretable reasoning, symbolic approaches encountered challenges handling data-driven learning and uncertainty. This limitation catalyzed interest in statistical and connectionist methods. Although neural networks, inspired by biological models, were explored from the late 1950s onward, early perceptrons suffered from limited expressive power [9], and practical training of multi-layer networks was hampered by inadequate algorithms and computational resources. These difficulties led to cycles of optimism followed by setbacks, known as the "AI Winters," during which neural research faced diminished funding and attention [10].

The resurgence of neural networks in the 1980s and 1990s, driven notably by Geoffrey Hinton's work on backpropagation and distributed representations [11, 12], demonstrated the feasibility of training deep architectures capable of modeling complex data. The explosion of large datasets and GPU computing in the 2000s accelerated progress, resulting in breakthroughs such as convolutional neural networks (CNNs) for computer vision [13] and advances in speech recognition—marking the deep learning revolution.

In natural language processing, recurrent neural networks (RNNs) and their variants like LSTMs initially dominated due to their sequence modeling capabilities but struggled with long-range dependencies and parallelization. The introduction of the Transformer architecture [14] revolutionized the field by replacing recurrence with self-attention, enabling efficient training on massive datasets while capturing global context more effectively. This architecture paved the way for large-scale pretrained language models such as GPT [15, 16], which transformed NLP through their ability to generate coherent, contextually rich text.

A pivotal advancement was the scaling of these Transformer-based models, guided by empirical scaling laws [17] that predicted consistent performance gains with increased model size and data, culminating in GPT-3 and the advent of few-shot prompting [18]. To better align these powerful models with human intent, reinforcement learning from human feedback (RLHF) was introduced [19] and further refined through summarization [20], instruction tuning exemplified by InstructGPT [21], and Constitutional AI [22]. Recent advances such as chain-of-thought prompting [23] and multitask instruction tuning [24] have significantly enhanced models' reasoning abilities.

Amid these breakthroughs, there has been a renewed interest in reconciling symbolic and neural paradigms—especially for autonomous agents capable of reasoning, reflection, and tool use [25]. Hybrid architectures aim to combine the explicit structure and manipulability of symbolic systems with the adaptive, data-driven learning of neural models.

Building on this trajectory, our work introduces a novel framework embedding large language models within a Lisp-based REPL environment. This approach seeks to empower self-programming agents with persistent memory and dynamic tool-building capabilities, harnessing Lisp's symbolic metaprogramming expressiveness alongside the generative and reflective strengths of modern LLMs.

3 Why Lisp?

Lisp is a compelling choice for integration with language models due to its syntactic simplicity, semantic power, and historical alignment with AI. Its hallmark feature—homoiconicity, or the uniform representation of code and data—naturally complements the capabilities of large language models, which benefit from easily parseable and manipulable structures. This property supports powerful metaprogramming and self-modifying constructs that are more cumbersome in other languages.

Lisp's minimalist, consistent syntax further enhances accessibility. Unlike modern languages with complex and irregular grammars, Lisp's uniform parenthesized expressions reduce both cognitive and computational load—allowing language models to focus more on semantic content than on syntactic correctness.

Historically, Lisp has been central to symbolic AI, with a legacy of systems, idioms, and literature focused on reasoning, language understanding, and knowledge representation. It remains one of the most expressive tools for modeling intelligent behavior—making it ideal for hybrid systems that blend symbolic and statistical reasoning.

A key advantage of Lisp is its extensibility. Macros, first-class symbols, and dynamic redefinition allow both human programmers and AI agents to evolve the language itself. This makes Lisp an ideal substrate for interactive programming environments in which LLMs are not just passive generators of code, but active participants in program construction and refinement.

Finally, Lisp benefits from a mature ecosystem of tools and documentation. These resources are vital for language models, which rely on large textual corpora to learn programming patterns. Well-documented and idiomatic Lisp code increases the reliability and usefulness of model-generated output—making Lisp not only a theoretical match but a pragmatic one as well.

3.1 Why Common Lisp?

Among Lisp dialects, Common Lisp stands out for its balance of theoretical rigor and practical power. Unlike Scheme, which favors minimalism, Common Lisp addresses real-world engineering needs with a rich standard library, a powerful object system (CLOS), advanced error-handling (via its condition system), and built-in support for concurrency and numerical computation.

Its macro system is particularly notable: beyond syntactic sugar, it enables full language extension. This is crucial in LLM contexts, where models can help create new abstractions or domain-specific languages dynamically—acting not just as code generators but as co-designers of language features.

Common Lisp also supports interactive development via the Read-Eval-Print Loop (REPL), facilitating incremental programming and continuous interaction—an ideal setting for LLM-assisted coding. The ability to modify the environment at runtime, redefine functions, and evolve systems without restarting supports the development of persistent, adaptive agents.

In short, Common Lisp combines theoretical elegance with engineering pragmatism. It is uniquely suited to serve as the foundation for AI-augmented programming environments, where language models are integrated not only as external tools, but as embedded reasoning components within a dynamic, extensible system.

4 Proposed System Architecture

The proposed system architecture enables the integration of language models with a live Lisp environment for real-time symbolic computation and interaction. It consists of three main components: a language model backend, a middleware layer, and a persistent Lisp REPL. Together, these components allow the model to reason, evaluate code, and continue generation based on live results from the Lisp runtime.

At the core of the system is the *language model backend*, responsible for processing user inputs and generating natural language responses that may include embedded Lisp code. The backend can be powered by either a local model (e.g., via Ollama) or a remote API (e.g., OpenAI's GPT). During generation, the language model may include Lisp expressions wrapped in a special tag format, such as `<lisp>...</lisp>`. This tag serves a function similar to internal tags like `<thinking>`, but instead of denoting a private reasoning process, it explicitly contains executable Lisp code.

The *middleware layer* operates as a stream-aware proxy that intercepts the output tokens generated by the language model. When it detects a `<lisp>...</lisp>` block, the middleware pauses the generation process, extracts the enclosed Lisp code, and forwards it to a running Lisp interpreter for evaluation. Once the evaluation is complete, the resulting value is captured and inserted back into the generation stream in place of the original tag. Generation then resumes as if the result had been part of the model's initial output. This strategy allows for dynamic, runtime computation and context-aware elaboration without disrupting the overall flow of interaction.

The final component is a *persistent Lisp REPL*, such as an SBCL instance, which maintains long-term program state and enables live code execution. This REPL holds definitions across turns, allowing functions, variables, and environments to persist and evolve. It also supports introspection, macro expansion, and dynamic redefinition, features that empower both symbolic reasoning and metaprogramming in collaboration with the language model.

Together, these three components create a hybrid symbolic-neural system in which the language model is not merely a static generator of code but an active participant in a live programming environment. The use of structured tags like `<lisp>` ensures a clean separation between natural language and executable code, enabling robust middleware orchestration and traceable interactions between text generation and symbolic evaluation.

5 Capabilities and Benefits

Integrating a live Lisp environment with a language model yields several powerful capabilities that extend beyond simple code generation. Chief among these is the ability to construct and retain stateful tools. Unlike ephemeral function calls or isolated code snippets, definitions in Lisp persist across invocations within the same REPL session. This allows the language model to accumulate a growing library of functions, macros, and utilities over time—effectively constructing an evolving toolkit that supports more complex behaviors and richer interactions as the session progresses.

A second benefit lies in Lisp's reflective nature. The language's introspective design enables the model to inspect its current environment dynamically. It can query available functions, examine their source code, and redefine procedures as needed. This supports self-aware generation flows in which the model not only writes code but reasons about its structure and behavior, debugging and refining it in real time.

Lisp also excels in metaprogramming. Through its powerful macro system and support for higher-order functions, the language enables the model to define domain-specific languages (DSLs) or control structures that are customized to its tasks. This metaprogrammatic flexibility empowers the model to shape the environment to its own needs, encapsulating complex patterns into reusable forms and adapting its interaction style accordingly.

One particularly forward-looking capability is the model's potential for generative self-extension.

By leveraging Lisp's ability to modify its own structure and behavior, the model may begin to construct generation pipelines, reusable routines, or wrappers to scaffold its own reasoning. While still speculative, this line of exploration offers a compelling research direction: the development of adaptive systems capable of modifying not just their outputs, but the strategies by which they think and communicate.

Finally, the incorporation of a persistent Lisp REPL provides a natural interface for enhancing reinforcement learning techniques focused on reasoning. The live, programmable environment acts as an external tool through which the model can test hypotheses, debug logic, and iteratively refine its strategies. By providing consistent feedback through concrete evaluations, the REPL enables learning algorithms to more effectively shape the model's internal policy. This integration of symbolic tooling with reinforcement learning opens new avenues for developing sophisticated and grounded thought processes in language models.

6 Use Cases

The integration of language models with a live Lisp environment enables a broad range of compelling applications across domains that require symbolic reasoning and dynamic tool construction. In scientific computing, the system can support symbolic algebra, equation solving, and the development of reusable computational tools tailored to specific experimental workflows. As an interactive programming assistant, the model can incrementally build its own toolkit during a session, creating and refining functions or macros as needed. Within agent frameworks, the architecture supports delegation of complex tasks to Lisp-defined subroutines or the orchestration of multiple agents through structured coordination. Additionally, the model can function as a language-to-logic translator, dynamically constructing interpreters that transform natural language instructions into executable code, offering a pathway toward more robust and adaptable natural language interfaces.

7 Safety Considerations

While the integration of a language model with a persistent Lisp environment offers powerful capabilities, it also safety concerns that have to be addressed. Allowing a language model to execute arbitrary code in a live environment introduces the risk of unsafe operations, including potential system access or manipulation of sensitive resources. To mitigate this risks, the system should be deployed in an isolated execution environment, such as a sandboxed container, where potentially unsafe functions can be restricted or controlled. Careful curation of the runtime environment, along with monitoring and access controls, is essential to maintain security while enabling dynamic interaction.

8 Conclusion

We have introduced a conceptual design for augmenting language models with a persistent Lisp REPL, enabling dynamic tool construction, reflective reasoning, and structured programmatic interaction. This persistent environment not only supports retention of function and state definitions over time but can also be extended to integrate access to external tools such as search engines, embedding encoders, and other utilities—thereby enhancing memory persistence and contextual grounding. Although this design remains conceptual and has yet to be implemented or empirically validated, it provides a modular, safe, and extensible foundation rooted in established programming paradigms. By bridging symbolic programming with neural language models, this approach lays the groundwork for future research in self-extending AI systems and hybrid architectures that leverage complementary strengths to build more interactive and adaptable LLM tools.

9 Future Work

9.1 Implementation Roadmap

A proof-of-concept implementation should focus on three foundational components: a streaming middleware capable of detecting and parsing `<lisp>` tags in real time; a persistent SBCL (Steel Bank Common Lisp) REPL with robust session and memory management; and basic sandboxing to ensure safe execution of dynamically generated code. Initial evaluations should assess the persistence and reliability of tools across conversational turns, as well as the performance trade-offs introduced by this symbolic integration relative to conventional function-calling approaches in LLM pipelines.

9.2 Research Directions

Several research avenues emerge from this framework. These include optimizing the streaming middleware pipeline for low-latency symbolic interaction, constructing benchmarks that evaluate hybrid symbolic-neural reasoning capabilities, and potentially extending the system to support other programmable languages beyond Lisp. Investigating emergent behaviors in self-modifying or recursively generated code could provide insights into meta-level reasoning dynamics. Moreover, safety analysis must include adversarial robustness, sandbox evasion attempts, and resource exhaustion scenarios to ensure predictable and controlled execution in open-ended environments.

9.3 Integration and Scalability

Future development should consider integration with existing large language model ecosystems. Scaling to distributed multi-agent environments may enable shared symbolic workspaces where agents coordinate tool construction and reasoning tasks. Open research questions include identifying the scalability limits of persistent symbolic state, developing mechanisms for meta-learning in tool generation and reuse, and implementing effective safety protocols as agents gain greater autonomy and capacity for reflection.

Acknowledgements

The author gratefully acknowledges the use of AI-assisted tools that played a pivotal role in shaping the ideas and presentation of this paper. In particular, OpenAI, DeepSeek, and Anthropic were invaluable for refining arguments, exploring alternative formulations, and improving clarity. These tools functioned as dynamic collaborators throughout the writing process.

The author also extends appreciation to the broader open-source community, whose contributions—spanning software, libraries, documentation, and shared knowledge—provided essential infrastructure and inspiration. This work builds upon the collective efforts of countless developers, researchers, and practitioners dedicated to open and accessible innovation.

References

[1] John McCarthy. Programs with common sense. Technical report, MIT, 1959. Presented at the Teddington Conference on Mechanisation of Thought Processes.

[2] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[3] Terry Winograd. Procedures as a representation for data in a computer program for understanding natural language. Technical Report AI-TR-235, MIT Artificial Intelligence Laboratory, 1971.

[4] Marvin Minsky. A framework for representing knowledge. Technical Report Memo 306, MIT AI Lab, 1974.

[5] Henk P Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

[6] Chris Hankin. An introduction to lambda calculi for computer scientists. *Electronic Notes in Theoretical Computer Science*, 125:1–57, 2005.

[7] William R Swartout. Review of: Rule-based expert systems: The mycin experiments of the stanford heuristic programming project. *Artificial Intelligence*, 1985. Book review of Buchanan and Shortliffe (1984), Addison-Wesley.

[8] Guy L Steele. *Common LISP: The Language.* Digital Press, 1990.

[9] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[10] Hubert L Dreyfus. *What computers still can't do: A critique of artificial reason.* MIT press, 1992.

[11] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[12] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[15] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. Technical report, OpenAI, 2018. OpenAI Technical Report.

[16] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):1–9, 2019.

[17] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[18] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[19] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in Neural Information Processing Systems*, 30, 2017.

[20] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020.

[21] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[22] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

[23] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[24] Victor Sanh, Albert Webson, Colin Raffel, Stephen H Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207*, 2021.

[25] Haoyi Xiong, Zhiyuan Wang, Xuhong Li, Jiang Bian, Zeke Xie, Shahid Mumtaz, Anwer Al-Dulaimi, and Laura E Barnes. Converging paradigms: The synergy of symbolic and connectionist ai in llm-empowered autonomous agents. *arXiv preprint arXiv:2407.08516*, 2024.