Accessible Musical ALife Through LLM Co-Creation

Michael Clemens
michael.clemens@outlook.com
Independent Researcher
Chesterton, IN, USA

Piotr Walas
Walas.piotr@outlook.com
Independent Researcher
Warsaw, Poland

Victor Shepardson
victor.shepardson@gmail.com
University of Iceland
Reykjavík, Iceland

Jack Armitage
jack.d.armitage@gmail.com
University of Iceland
Reykjavík, Iceland

Abstract

GPU-accelerated artificial life (ALife) simulations offer rich possibilities for musical expression, but building them demands programming fluency that excludes many musicians and sound artists. We present a natural language interface that translates behavioral descriptions into executable particle simulations with integrated Open Sound Control output and SuperCollider companion patches. A preliminary evaluation across three frontier models suggests that structured context engineering is necessary for reliable code synthesis, while extensive documentation alone yields no improvement over unassisted generation. The pipeline's value lies in overcoming the *cold start* of domain specific languages, producing a first sketch for the artist to refine.

CCS Concepts

• **Applied computing** → **Sound and music computing**; • **Computing methodologies** → **Artificial life**; Natural language processing; • **Human-centered computing** → *Interactive systems and tools.*

Keywords

Artificial Life, Musical Interface, LLM Code Synthesis, OSC, Sonification, Context Engineering

1 Introduction

Artificial Life (ALife), the synthesis of life-like processes in non-biological media [13], offers musicians an alternative to parameter mapping: rather than automating control curves, the artist tends an ecosystem of agents whose emergent behaviors drive composition [21, 22]. GPU-accelerated ALife simulation, however, demands fluency in kernel compilation, parallel data structures, and framework-specific constraints [10], creating a barrier for musicians whose expertise lies elsewhere [4, 23].

Large language models (LLMs) appear to offer a bridge, translating behavioral descriptions into executable code. In practice, even frontier models fail to generate valid code for specialized GPU frameworks regardless of how much documentation they receive [14]. We present a natural language interface that addresses this gap, generating GPU-accelerated particle simulations with integrated Open Sound Control (OSC) [27] output and SuperCollider [16] companion patches.

This work is licensed under a Creative Commons Attribution 4.0 International License.
NIME '26, June 23–26, 2026, London, UK
© 2026 Copyright held by the owner/author(s).

Our preliminary evaluation suggests that structured context engineering (decomposing prompts and selecting targeted domain knowledge) is necessary for reliable synthesis, while extensive documentation alone does not help. The same pipeline generalizes across frontier models without model-specific tuning, suggesting that targeting smaller open-weight models may paradoxically require establishing such pipelines on frontier models first. The pipeline's principal value lies in overcoming the *cold start* of domain specific languages (DSLs), producing a bespoke first sketch the musician refines through conversation. Source code and evaluation scripts are available at https://github.com/mclemcrew/tolvera.

2 Related Work

Wekinator [7] established that interactive machine learning can be made accessible to musicians without machine learning expertise, a precedent our work extends to GPU-accelerated ALife. Creative coding environments such as Processing [20], TidalCycles [17], and LLM-powered tools such as Spellburst [2] have further lowered the barrier to artistic computation [12]. NIME practice, however, remains predicated on domain-specific languages (e.g., SuperCollider [16], ChucK [26], Faust [19], CSound [25]), for which LLMs have significantly less training data than general-purpose languages [11]. GPU frameworks such as Taichi [10] compound this, requiring kernel compilation and parallel data layout that creative coding environments do not abstract away.

While LLMs demonstrate strong synthesis for general-purpose languages [6], domain-specific frameworks pose distinct challenges general models cannot reliably handle [8]. Zhang et al. [28] benchmark LLM generation for audio programming and find

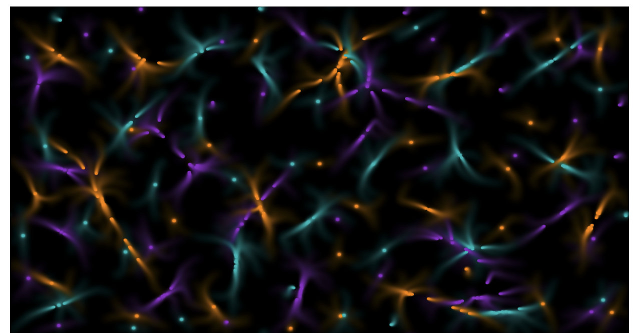

Figure 1: Output of the illustrative example. Three species of flocking particles with diffusion trails. Each species' leader particle streams position and velocity data via OSC to a dedicated SuperCollider SynthDef, producing a three-voice evolving texture driven by spatial movement.

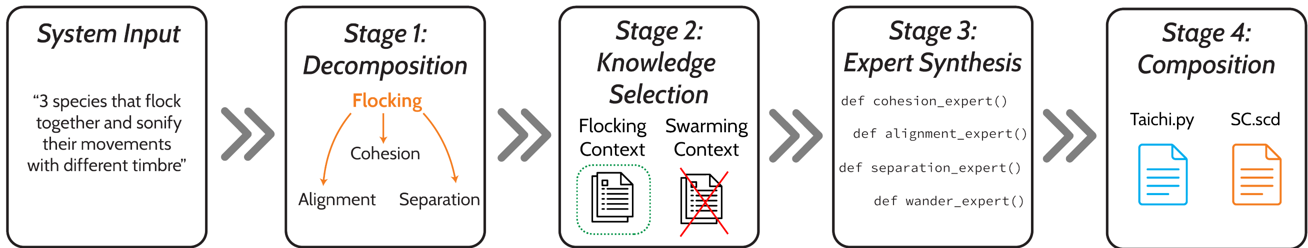

Figure 2: System pipeline. A natural language description is decomposed into atomic behavioral components, enriched with dynamically-selected domain context, synthesized as individual GPU expert functions, and composed into an executable sketch with optional OSC output and SuperCollider companion scripts.

that structured approaches yield more correct output, corroborating our findings. Promising mitigations include type-constrained decoding [18], monitor-guided decoding that enables smaller models to match frontier ones [1], and low-rank adaptation on curated DSL corpora [24].

At the intersection of ALife and music, Lucas et al. [15] demonstrated interactive sonification of swarmalator systems at NIME, treating swarm dynamics as a real-time sound source. We extend this direction by using LLMs to translate natural language *intent* into executable GPU code driving both visual and sonic output.

3 System Overview

To address these technical barriers, we extend Tölvera [3], an ALife framework for composable agents, with a system that translates natural language behavior descriptions into executable particle simulations via Taichi [10]. The pipeline is carried out in four stages. First, an LLM decomposes the user's description into atomic behavioral components, identifying species, state variables, and musical intent. A second LLM call selects relevant domain knowledge modules (typically 3–5), acting as a lightweight retrieval step. Each component is then synthesized as an individual GPU kernel function, constrained by the selected context and typing rules. We term these synthesized functions *experts*, inspired by the Product of Experts formulation [9], as the final behavior emerges from their combination. Finally, Jinja2[1] templates compose the experts into a complete executable sketch.

3.1 Sonification & OSC Integration

When a user's prompt contains musical keywords (e.g., "sonify," "sound," "rhythm," "musical"), the system generates OSC mappings alongside the visual simulation. A dedicated module selects per-species metrics (position, velocity, density, separation), and emits smoothing kernels for stable real-time control. Because OSC is host-agnostic, any OSC-capable environment can consume these streams; we provide a SuperCollider companion patch as a reference mapping (position to pitch over 80–3000 Hz exponentially, velocity to amplitude, density to filter cutoff).

3.2 Illustrative Example

To illustrate the pipeline, we prompted the system with "*three species of particles that flock together and sonify their movements with different timbres.*" The decomposer identified four expert types (separation, cohesion, alignment, wander), the context selector retrieved the flocking and OSC modules, and each expert was synthesized individually before composition into a complete sketch streaming three independent OSC channels (one per species).

[1]https://jinja.palletsprojects.com/en/stable/

We routed these to a SuperCollider companion patch with three SynthDefs (saw, FM, pulse) mapping position to pitch and velocity to amplitude.

4 Evaluation

We compared the context-engineered pipeline against unassisted frontier models on 8 prompts with binary pass/fail. The pass criterion (error-free execution for 12 seconds) checks syntactic and runtime correctness only, not behavioral fidelity or musical utility; user studies addressing those dimensions are future work. Prompts span four difficulty levels (T1: single-particle physics, T2: pairwise interactions, T3: multi-species behaviors, T4: complex multi-component systems), each containing 2 prompts with musical/OSC intent. Given the sample size, results are illustrative of qualitative differences between conditions rather than performance claims.

4.1 Benchmark Results

Table 1: Execution pass rate by model, condition, and difficulty tier. Each cell shows passes out of 2 prompts. All prompts include musical/OSC intent.

Model	Condition	T1	T2	T3	T4	Total
Sonnet 4.5	Zero-shot	0/2	0/2	0/2	0/2	0/8
	Full codebase	0/2	0/2	0/2	0/2	0/8
	Pipeline	2/2	1/2	2/2	1/2	**6/8**
Opus 4.6	Zero-shot	0/2	0/2	0/2	0/2	0/8
	Full codebase	0/2	0/2	0/2	0/2	0/8
	Pipeline	1/2	1/2	2/2	1/2	**5/8**
DeepSeek v3	Zero-shot	0/2	0/2	0/2	0/2	0/8
	Full codebase	0/2	0/2	0/2	0/2	0/8
	Pipeline	1/2	1/2	1/2	1/2	**4/8**

We tested three conditions: (A) zero-shot with minimal framework context, (B) full codebase documentation as system context, and (C) our context-engineered pipeline. Table 1 shows conditions A and B achieve 0% execution across all models, while the pipeline produces runnable sketches at 75% (Sonnet), 62% (Opus), and 50% (DeepSeek). Dominant failures without the pipeline include hallucinated APIs (e.g., `osc.create_client()`) and GPU-specific syntax violations. Condition B offered no measurable improvement over zero-shot, possibly because irrelevant API surface introduces noise rather than informing generation. Selective context (3–5 targeted modules) outperformed providing all

modules at once, suggesting that *how* context is structured matters more than *how much*.

4.2 Frontier vs. Local Models

Local models offer openness and offline capability but failed to satisfy Taichi's strict typing and kernel constraints even within our pipeline, often hallucinating standard Python functions (e.g., `random.random()`) inside GPU kernels where framework-specific intrinsics (e.g., `ti.random()`) are required. Frontier models succeeded only when constrained by the pipeline, suggesting pipeline architecture may matter more than raw model capability for this class of task.

5 Discussion

5.1 Iteration vs. Generation

A recurring challenge in algorithmic composition is starting from zero. LLMs are better at *iterating* on existing code than at *creating* from a blank slate; in our informal testing, all models could make meaningful changes to an existing sketch, but generation from nothing almost always failed. Specialized GPU frameworks intensify the gap, since iteration presupposes producing the boilerplate to get the first pixel on screen. The system therefore functions less as a composer than as a means of overcoming the activation energy needed to initialize a complex ALife sketch, so the musician can begin refining immediately. Refining generated code still benefits from programming familiarity, but the distance between modifying a working sketch and writing one from scratch is substantially smaller. Our correctness-oriented evaluation also misses an important dimension of creative value: each generation introduces different details and small deviations that can themselves be artistically productive, echoing creative coding's tradition of serendipitous bugs yielding unexpected directions [5].

5.2 Constrained Generation Beats Larger Models

We initially prioritized openness, but smaller models required prompt optimization that the rapid release cycle of new models frequently rendered obsolete. Comprehensive framework documentation did not yield measurable improvement over unassisted generation. In contrast, a pipeline defined by prompt decomposition and context selection produced working sketches across frontier models without model-specific tuning. Our small sample suggests reliability may stem from architectural context engineering rather than raw model capability; future work targeting smaller models will likely require establishing baselines on frontier models first.

5.3 Accessibility & Openness

Pairing visual simulations with a SuperCollider companion lowers the barrier for practitioners without GPU programming skills. A tension remains between capability and openness: frontier models are closed-source and commercially hosted, while open-weight alternatives that run locally and can be inspected by the community cannot yet satisfy the strict constraints of GPU kernel synthesis. Since NIME practice broadly relies on DSLs, this *openness-capability gap* extends beyond our system [11]. Constrained decoding and domain-specific fine-tuning (Section 2) offer promising directions for closing it while preserving artist autonomy.

6 Ethics Statement

Large language models are both the subject and a tool of this research. The pipeline evaluates frontier models (Sonnet 4.5, Opus 4.6, DeepSeek v3) as components of a code synthesis system, and all generated outputs are clearly identified as such throughout the paper. Original drafting, argumentation, and analysis were completed by the human authors; Claude Code and Gemini were subsequently used as reviewing tools to surface poor phrasing, weak arguments, and logical gaps, with all final editorial decisions remaining with the human authors.

The evaluation is fully automated, scored by subprocess execution, and involved no human participants; user studies of creative affordances and accessibility are future work and will be conducted under institutional ethics review. We designed the system to lower the barrier to GPU-accelerated musical ALife for practitioners whose expertise lies in composition rather than kernel programming, and the source code and evaluation scripts are released under an open-source license to support community access and reproducibility.

We acknowledge the computational costs of this work, which include repeated LLM inference calls to commercially hosted models during development and evaluation, as well as GPU execution of Taichi simulations. Individual sketch generation is modest, but iterative benchmarking across multiple models and conditions accumulates non-trivial energy expenditure. Finally, as discussed in Section 5.3, a tension exists between the capability of closed-source frontier models and the values of openness and artist autonomy that inform NIME practice; we view constrained decoding and domain-specific fine-tuning as promising directions for closing this gap.

7 Conclusion

We have presented a natural language interface for musical ALife co-creation. Our preliminary results suggest structured context engineering is necessary for reliable synthesis in specialized creative frameworks, and that the pipeline's value lies in overcoming the *cold start* of specialized syntax. Larger-scale evaluation is needed, but these observations may be of use to others building LLM-powered tools for creative communities where the gap between artistic intent and technical implementation remains wide.

Acknowledgments

We thank the Tölvera community and Google Summer of Code mentors for feedback during development.

References

[1] Lakshya A. Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*.

[2] Tyler Angert, Miroslav Ivan Suzara, Jenny Han, Christopher Lawrence Pondoc, and Hariharan Subramonyam. 2023. Spellburst: A Node-based Interface for Exploratory Creative Coding with Natural Language Prompts. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. ACM. https://doi.org/10.1145/3586183.3606719

[3] Jack Armitage, Victor Shepardson, and Thor Magnusson. 2024. Tölvera: Composing With Basal Agencies. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, S M Astrid Bin and Courtney N. Reed (Eds.). Utrecht, Netherlands, Article 42, 10 pages. https://doi.org/10.5281/zenodo.13904854

[4] Greg Burlet and Abram Hindle. 2015. An Empirical Study of End-User Programmers in the Computer Music Community. In *Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 292–302.

[5] Kim Cascone. 2000. The Aesthetics of Failure: "Post-Digital" Tendencies in Contemporary Computer Music. *Computer Music Journal* 24, 4 (2000), 12–18. https://doi.org/10.1162/014892600559489

[6] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[7] Rebecca Fiebrink and Perry R. Cook. 2010. The Wekinator: A System for Real-Time, Interactive Machine Learning in Music. In *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR 2010), Late-Breaking/Demo Session*. Utrecht, Netherlands.

[8] Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. 2025. On the Effectiveness of Large Language Models in Domain-Specific Code Generation. *ACM Transactions on Software Engineering and Methodology* 34, 3 (2025), 78. https://doi.org/10.1145/3697012

[9] Geoffrey E. Hinton. 2002. Training Products of Experts by Minimizing Contrastive Divergence. *Neural Computation* 14, 8 (2002), 1771–1800. https://doi.org/10.1162/089976602760128018

[10] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–16. https://doi.org/10.1145/3355089.3356506

[11] Sathvik Joel, Jie JW Wu, and Fatemeh Fard. 2024. A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages. *arXiv preprint arXiv:2410.03981* (2024).

[12] Stephen James Krol, Maria Teresa Llano Rodriguez, and Miguel Loor Paredes. 2025. Exploring the Needs of Practising Musicians in Co-Creative AI through Co-Design. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3706598.3713894

[13] Christopher G. Langton. 1989. Artificial Life. In *Artificial Life: Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems*. Addison-Wesley, Redwood City, CA, 1–47.

[14] Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, WangHaojie WangHaojie, Jianrong Wang, Xu Han, et al. 2025. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*. 23053–23066.

[15] Pedro P. Lucas, Stefano Fasciani, Alexander Szorkovszky, and Kyrre Glette. 2024. Interactive Sonification of 3D Swarmalators. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2024)*. Utrecht, Netherlands, 252–260. https://doi.org/10.5281/zenodo.13904846

[16] James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal* 26, 4 (2002), 61–68. https://doi.org/10.1162/014892602320991383

[17] Alex McLean. 2014. Making Programming Languages to Dance To: Live Coding with Tidal. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling & Design*. ACM, 63–70.

[18] Niels Mündler, Jingxuan He, Haobin Wang, Koushik Sen, Dawn Song, and Martin Vechev. 2025. Type-Constrained Code Generation with Language Models. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025). https://doi.org/10.1145/3729299

[19] Yann Orlarey, Dominique Fober, and Stéphane Letz. 2009. FAUST: An Efficient Functional Approach to DSP Programming. In *New Computational Paradigms for Computer Music*. Editions DELATOUR FRANCE, 65–96.

[20] Casey Reas and Ben Fry. 2007. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press.

[21] Craig W. Reynolds. 1987. Flocks, Herds and Schools: A Distributed Behavioral Model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87)*. ACM, 25–34. https://doi.org/10.1145/37401.37406

[22] Hiroki Sayama. 2009. Swarm Chemistry. *Artificial Life* 15, 1 (2009), 105–114. https://doi.org/10.1162/artl.2009.15.1.15107

[23] Andrew Telichan-Phillips and Daniel Marchwinski. 2024. Hidden Influences in Music Technology: An Approach to Coding Practice. In *Music, the Avant-Garde, and Counterculture: Invisible Republics*. Springer, 189–200.

[24] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. VeriGen: A Large Language Model for Verilog Code Generation. *ACM Transactions on Design Automation of Electronic Systems* 29, 3 (2024), 46. https://doi.org/10.1145/3643681

[25] Barry L. Vercoe. 1986. *Csound: A Manual for the Audio Processing System*. Technical Report. MIT Media Laboratory, Cambridge, MA.

[26] Ge Wang and Perry R. Cook. 2003. ChucK: A Concurrent, On-the-fly, Audio Programming Language. In *Proceedings of the International Computer Music Conference (ICMC)*. Singapore.

[27] Matthew Wright and Adrian Freed. 1997. Open Sound Control: A New Protocol for Communicating with Sound Synthesizers. In *Proceedings of the International Computer Music Conference (ICMC)*. Thessaloniki, Greece, 101–104.

[28] William Zhang, Maria Leon, Ryan Xu, et al. 2024. Benchmarking LLM Code Generation for Audio Programming with Visual Dataflow Languages. *arXiv preprint arXiv:2409.00856* (2024).