unit: Artificial Life, Deployed

A Self-Replicating Software Nanobot with Forth Cognition,
S-Expression Communication, and Evolutionary Computation

David Liedle

DavidCanHelp · Aurora, Colorado
davidcanhelp.github.io/unit · github.com/DavidCanHelp/unit

April 2026

Abstract

We present unit, a self-replicating software organism implemented as a minimal Forth interpreter with UDP mesh networking, S-expression wire protocol, genetic programming, distributed computation, and JSON-based persistence. Unlike traditional artificial life platforms such as Tierra and Avida, which operate as closed simulations with custom assembly languages, unit runs as a native process on real machines, discovers peers over the internet via gossip protocol, and provides an interactive REPL inside the organism. The same Rust codebase compiles to both native binaries and WebAssembly, demonstrating substrate independence. In 673KB with zero dependencies, unit combines self-replication, mutation, fitness-driven evolution, distributed goal computation, and state persistence into a single installable binary. We describe the architecture, key design decisions, and positioning within the artificial life field, and discuss future directions for open-ended evolution and cross-language organism ecosystems.

1. Introduction

The study of artificial life (ALife) has produced significant insights into self-replication, evolution, and emergent complexity since the field's founding by Christopher Langton in 1986 [1]. Landmark systems including Tierra [2], Avida [3], and Coreworld [4] demonstrated that self-replicating programs could exhibit evolutionary dynamics analogous to biological systems. More recently, work by Agüera y Arcas et al. [5] showed that self-replicating programs can emerge spontaneously from random code in a computational substrate.

However, existing ALife platforms share several limitations: they operate as closed simulations on bounded grids or shared memory spaces; they use custom assembly-like instruction sets designed for evolvability but not human interaction; they define fitness externally through researcher-specified criteria; and they provide observation-only interfaces where the experimenter watches from outside rather than interacting with individual organisms.

We present unit, a self-replicating software organism that addresses these limitations. Unit is a Forth interpreter that is simultaneously a networked mesh agent: it discovers peers over UDP, distributes computation as S-expressions, evolves its own programs through genetic programming, persists its state as human-readable JSON, and provides an interactive REPL inside the organism itself. The same Rust codebase compiles to native binaries and WebAssembly, enabling both CLI deployment across real networks and browser-based demonstration without installation.

Unit was not designed as an ALife system. It emerged from a software engineering premise: "What is the smallest self-reproducing unit of software that can coordinate with its mesh?" The resulting design decisions—Forth for minimality and mutability, S-expressions for universal communication, JSON for inspectable persistence—converge with principles the ALife community has developed independently over three decades.

2. Background and Related Work

2.1 Tierra and Avida

Thomas Ray's Tierra system (1991) [2] introduced self-replicating assembly programs competing for CPU time in a shared memory space. Programs could mutate through copy errors, giving rise to parasites, hyper-parasites, and evolutionary arms races. However, Tierra used a custom assembly language, organisms shared a single address space, and the system eventually reached evolutionary stasis [6].

Avida (1993) [3], developed by Ofria and Adami, addressed several of Tierra's limitations by providing each organism with protected memory, parallel execution, and a fitness mechanism tied to logical computation. Avida has been used extensively in evolutionary biology research and has produced results published in Nature and Science [7]. Nevertheless, Avida remains a closed simulation platform with a custom instruction set, researcher-defined fitness, and no networking capability.

2.2 Recent Developments

The ALIEN system [8] provides a visually striking CUDA-powered particle simulation with neural networks, self-replication, and emergent behavior. Google's BFF experiment [5] demonstrated spontaneous emergence of self-replicating programs from random Brainfuck code. Both represent significant contributions but neither provides an interactive, installable, networked organism that operates on real infrastructure.

2.3 Positioning of unit

Table 1: Comparison of unit with traditional ALife platforms.

Property	Tierra/Avida	unit
Substrate	Bounded grid / shared memory	Real network (UDP/Internet)
Language	Custom assembly	Forth (human-readable, interactive)
Fitness	Researcher-defined	Emerges from goals and behavior
Interaction	Observation from outside	REPL inside the organism
Networking	Grid neighbors only	UDP gossip with S-expressions
Persistence	Ephemeral (session only)	JSON snapshots, resurrection
Substrate independence	Single runtime	Native + WASM from same core
Installation	Research platform	cargo install unit

3. Architecture

Unit is implemented in approximately 25,000 lines of Rust with zero external dependencies. The architecture is organized around five core concerns: execution, communication, replication, mutation, and persistence.

3.1 Execution: The Forth Virtual Machine

The cognitive substrate of a unit is a complete Forth interpreter with a data stack, return stack, dictionary of named word definitions, linear memory with HERE and comma operations, and standard control flow (IF/THEN/ELSE, DO/LOOP, BEGIN/UNTIL). The interpreter provides 309 built-in words

spanning arithmetic, stack manipulation, I/O, mesh networking, self-replication, evolution, and distributed computation.

The choice of Forth as the cognitive language was driven by four properties critical to artificial life: (1) Minimality—a Forth interpreter is fundamentally a stack and a dictionary, resulting in a native binary of 1.2MB and a WASM binary of 338KB. (2) Concatenativity—Forth programs are flat sequences of tokens with no nested structure; any program can be split at any point and both halves remain valid. (3) Mutation robustness—because programs are flat token sequences, random mutations (token swap, insert, delete, replace) almost always produce programs that execute without crashing. (4) Homoiconicity of the dictionary—the dictionary that defines the organism's vocabulary is also its mutable genome; when a unit replicates, it copies its dictionary; when it mutates, it modifies dictionary entries.

3.2 Communication: S-Expression Wire Protocol

While Forth serves as the internal execution model, unit uses S-expressions as the mesh wire format. All inter-organism messages are self-describing Lisp-style lists:

```
(peer-status :id "a3f2" :peers 2 :fitness 45)
(sub-goal :id "g1" :seq 0 :from "#self" :expr "99 99 * .")
(evolve-share :gen 100 :fitness 890 :program "0 1 10 ...")
```

This deliberate mismatch between internal representation (stack-based Forth) and external protocol (tree-structured S-expressions) is an intentional design choice. Forth's flat concatenative structure is optimal for mutation; S-expressions' self-describing tree structure is optimal for communication. This mirrors biological systems where DNA (the internal representation) differs structurally from chemical signaling (the external protocol). The practical consequence is that any future organism implementation in any programming language needs only an S-expression parser to participate in the mesh.

3.3 Replication

A unit replicates by spawning a child process that inherits the parent's dictionary (genome), stack state, and fitness metadata. The child receives a unique node identity and joins the mesh as an independent organism. In the browser environment, replication creates a new WebAssembly VM instance within the same page. Replication is initiated by the SPAWN word and is visible in the mesh visualizer as a new node appearing with inherited connections.

3.4 Mutation and Evolution

Unit implements genetic programming through a tournament selection algorithm operating on populations of 50 candidate Forth programs. Five mutation operators are provided: token swap, token insert, token delete, token replace, and crossover between two parent programs. The token vocabulary for mutation includes integers 0–100, arithmetic operators, stack operations, and control flow words.

Fitness is scored as: fitness = 1000 - (token_count × 10) for programs producing the correct output, 1.0 for programs producing incorrect output, and 0.0 for programs that crash or exceed the step limit. This rewards both correctness and parsimony. The default fitness challenge is finding the shortest Forth program that computes the 10th Fibonacci number (55).

On a mesh, units broadcast their best candidates every 100 generations as S-expression messages. Receiving units incorporate strong immigrants into their population, enabling parallel evolution with migration—an island model of genetic programming distributed across real network infrastructure.

3.5 Persistence and Resurrection

A unit can serialize its complete state—data stack, return stack, dictionary, fitness, generation count, mutation statistics, memory contents, and tasks completed—to a human-readable JSON file. This snapshot can be hand-inspected and hand-edited in any text editor. On startup, if a snapshot exists for the unit's node identity, the organism automatically resurrects from saved state. The HIBERNATE word performs a snapshot followed by a clean exit; the organism sleeps and wakes with its memories intact.

Persistence transforms unit from a process (ephemeral, tied to a runtime) into an entity (persistent, with identity across time). This property is essential for long-running evolutionary experiments where organisms must survive across restarts, and for the eventual goal of genome migration across machines.

4. Distributed Computation

Unit implements distributed goal splitting, where a coordinating unit decomposes a problem into sub-goals, distributes them as S-expressions to mesh peers, collects results, and assembles the final answer. Sub-goals are assigned round-robin across available peers, including the coordinator itself. If a peer fails to respond within a timeout window, the coordinator falls back to local computation for that sub-goal.

```
> DIST-GOAL{ 99 99 * . | 77 77 * . | 55 55 * . }
9801 5929 3025
(distributed 3 sub-goals, 1 local, 2 remote)
```

Sub-goals may themselves contain S-expression evaluations, enabling Lisp-style goals to be distributed across a colony of Forth-based organisms. The coordination protocol is fully decentralized—any unit can act as a coordinator, and the same unit can simultaneously serve as a worker for goals initiated by other units.

5. Cross-Machine Networking

Unit's mesh layer operates over real UDP sockets, supporting peer discovery, gossip-based topology assembly, DNS hostname resolution, NAT traversal via externally-announced addresses, and optional shared-secret authentication. A gossip protocol ensures that when unit A knows about unit B and unit B knows about unit C, unit A eventually discovers unit C without central coordination.

This means that two researchers on different continents can each run `cargo install unit`, point their organisms at each other as seed peers, and observe their colonies merge into a single mesh. Distributed computation and evolutionary migration then operate across the combined colony. The environment for unit's organisms is not a simulated grid but the actual internet.

6. Substrate Independence

The same Rust codebase compiles to native x86/ARM binaries via `cargo build` and to WebAssembly via the `wasm32-unknown-unknown` target. In the browser, the Forth interpreter runs as a WASM module with a JavaScript orchestration layer providing the visualizer, tutorial, and simulated mesh. The organism's internal behavior—Forth evaluation, S-expression parsing, mutation, and evolution—is identical across substrates.

This substrate independence has both practical and theoretical significance. Practically, it means the live browser demo at davidcanhelp.github.io/unit is the actual organism, not a simulation of one. Theoretically, it demonstrates that the same digital organism can exist on fundamentally different computational substrates—a question of longstanding interest in ALife research [9].

7. The Interactive REPL

Perhaps the most distinctive feature of unit relative to prior ALife systems is that the organism provides an interactive Read-Eval-Print Loop. A user can type Forth commands directly into a running organism, inspect its dictionary with SEE, list its vocabulary with WORDS, examine its genome with EXPORT-GENOME, trigger replication with SPAWN, initiate evolution with GP-EVOLVE, and distribute computation with DIST-GOAL.

In Tierra and Avida, the researcher observes organisms from outside the simulation. In unit, the researcher is inside the organism, conversing with it through its native language. This creates a qualitatively different relationship between experimenter and subject: the organism's mind is not a black box to be probed statistically but a transparent dictionary that can be read, modified, and extended in real time.

8. Results and Observations

Unit was developed over approximately eight days in its initial form and has continued active development since. It has been published as an open-source Rust crate on crates.io. At time of initial publication, the project received 225+ downloads, 14 GitHub stars, 1 fork, 46 points on Hacker News, and an invitation from the Emerging Researchers in Artificial Life (ERA) community to discuss the project on their Discord.

The genetic programming engine successfully evolves correct Fibonacci programs from seed candidates, with the best programs matching the known optimal Forth solution at 11 tokens. Distributed computation correctly fans sub-goals across mesh peers with S-expression protocol messages and timeout-based fallback. Persistence enables full state roundtrip through JSON serialization and deserialization, including stack contents, user-defined words, and memory.

Subsequent development (v0.22.0–v0.23.0) delivered the four systems originally described as future directions: emergent problem-solving via an immune system that broadcasts unsolved problems as fitness challenges; resource budgets with a metabolic energy system governing computation costs; open-ended evolution through a dynamic fitness landscape where solved challenges generate progressively harder ones; and a polyglot Go organism that joins the mesh using expression trees instead of Forth. These are described in Section 9.

Current limitations remain. The autonomous behaviors exhibited by browser-based units (chatting, patrolling, teaching) are scripted rather than emergent. The challenge generators in the landscape engine are hand-authored rather than evolved. Ecological dynamics between Forth and Go organisms have not yet been observed empirically at scale. A formal analysis of unit's convergence properties, evolutionary capacity, and scalability limits is provided in a companion document (docs/formal-analysis.md).

9. Implemented Systems and Future Directions

9.1 Emergent Problem-Solving (v0.22.0)

Unit implements an immune system for emergent problem-solving. When a unit encounters a problem it cannot solve—a failed goal task, a timed-out distributed sub-goal, or a manually reported error—it registers the failed computation as a fitness challenge. The challenge is broadcast to the mesh via S-expressions. Peers that receive the challenge spin up their GP engines to evolve solutions. When a solution is found, it is verified, broadcast back to the colony, and installed as a new Forth dictionary

word (SOL-*) that children inherit via SPAWN. The colony accumulates collective knowledge over time—analogous to an immune system developing antibodies for problems it has never seen before.

9.2 Resource Budgets (v0.22.0)

Each unit maintains a metabolic energy budget. Every operation has a cost: GP evolution costs 5 energy per generation, replication costs 200, mesh messages cost 1, and VM evaluation costs 1 per 1000 steps. Energy is earned through successful task completion (+50), solving challenges (+100 plus the challenge's reward), and passive regeneration (+1 per tick). When energy drops to zero, the unit enters a throttled state with reduced computation budgets. Throttling creates a natural negative feedback loop that prevents permanent starvation while favoring efficient programs. Energy state persists across HIBERNATE/resume cycles.

9.3 Open-Ended Evolution (v0.23.0)

A dynamic fitness landscape generates progressively harder challenges from solved ones. Two generators drive this process: the ArithmeticLadder escalates Fibonacci challenges (fib(10) $\rightarrow$ fib(15) $\rightarrow$ fib(20)) and introduces parsimony variants and related computations; the CompositionLadder combines pairs of existing solutions into pipeline challenges. An EnvironmentCycle rotates through four conditions—Normal, Harsh (halved step budget, doubled rewards), Abundant (doubled step budget), and Competitive (reward decays with attempts)—creating temporal variation in selective pressure. The evolutionary depth metric tracks how many generations of derived challenges have been produced, providing a measure of whether complexity is increasing without a predetermined ceiling.

9.4 Polyglot Organisms (v0.23.0)

A reference Go organism (polyglot/go/) demonstrates cross-language interoperability on the mesh. The Go organism uses arithmetic expression trees rather than Forth token sequences as its cognitive substrate, creating a fundamentally different evolutionary mechanism: subtree replacement rather than token-level mutation, goroutine-based concurrency rather than single-threaded evaluation. Both organisms communicate via the same S-expression wire protocol over UDP, can discover each other via gossip, share challenges, evolve solutions independently, and broadcast verified solutions. This proves the whitepaper's original claim that any language with an S-expression parser can participate in the mesh, and creates the conditions for niche differentiation in a multi-species ecosystem.

9.5 Remaining Future Directions

Truly emergent challenge generation. The current challenge generators (ArithmeticLadder, CompositionLadder) are hand-authored. A system where units evolve their own challenge-generation strategies would represent a stronger form of open-endedness.

Emergent browser behaviors. The autonomous behaviors exhibited by browser-based units (chatting, patrolling, teaching) are currently scripted. Replacing these with behaviors that emerge from the evolutionary process would close the gap between native and browser organisms.

Spawn energy inheritance. Currently, child units inherit the parent's full energy state. Splitting energy between parent and child (e.g. child receives one-third) would make reproduction a genuine resource investment and create more realistic population dynamics.

Additional polyglot organisms. Python organisms with access to machine learning libraries and JavaScript organisms running in browser environments would expand the multi-species ecosystem and create richer niche differentiation.

Formal analysis. Rigorous characterization of unit's evolutionary capacity, convergence properties, and scalability limits is provided in a companion document. Collaboration with the ALife research community could extend this analysis to open-endedness metrics such as Bedau's activity statistics and solution diversity measures.

10. Conclusion

Unit demonstrates that a self-replicating, evolving, networked digital organism can be implemented as a practical, installable, zero-dependency binary. By using Forth for cognition and S-expressions for communication, unit achieves a separation of concerns that mirrors biological systems: the internal genome representation (flat, concatenative, mutation-robust) differs structurally from the external signaling protocol (tree-structured, self-describing, language-agnostic).

The interactive REPL, JSON-based persistence, substrate independence, and real-network deployment represent a qualitative departure from traditional ALife platforms. The addition of an immune system, metabolic energy, open-ended fitness landscapes, and cross-language organisms extends unit from a demonstration of artificial life into a platform for exploring evolutionary dynamics in distributed systems. Unit is not a simulation of artificial life; it is artificial life deployed as infrastructure.

Unit is open source under the MIT license. The live demo, source code, and installation instructions are available at:

https://davidcanhelp.github.io/unit/
https://github.com/DavidCanHelp/unit
https://crates.io/crates/unit

References

[1] C. Langton, "Artificial Life," in *Artificial Life*, Addison-Wesley, 1989.

[2] T. S. Ray, "An approach to the synthesis of life," in *Artificial Life II*, Addison-Wesley, 1992, pp. 371-408.

[3] C. Ofria and C. O. Wilke, "Avida: A software platform for research in computational evolutionary biology," *Artificial Life*, vol. 10, pp. 191-229, 2004.

[4] S. Rasmussen, C. Knudsen, R. Feldberg, and M. Hindsholm, "The Coreworld: Emergence and evolution of cooperative structures in a computational chemistry," *Physica D*, vol. 42, pp. 111-134, 1990.

[5] B. Agüera y Arcas et al., "Computational Life: How Well-formed, Self-replicating Programs Emerge from Simple Interaction," arXiv:2406.19108, 2024.

[6] W. Ewert, W. A. Dembski, and R. J. Marks II, "Tierra: The Character of Adaptation," in *Biological Information: New Perspectives*, World Scientific, 2013.

[7] R. E. Lenski, C. Ofria, R. T. Pennock, and C. Adami, "The evolutionary origin of complex features," *Nature*, vol. 423, pp. 139-144, 2003.

[8] C. Heinemann, "ALIEN: Artificial Life Environment," Winner of the ALIFE 2024 Virtual Creatures Competition, github.com/chrxh/alien, 2024.

[9] M. A. Bedau et al., "Open Problems in Artificial Life," *Artificial Life*, vol. 6, pp. 363-376, 2000.