

PERSISTENT RECURSIVE WORLDS ENABLE AUTONOMOUS SOFTWARE EVOLUTION

Beichen Huang Zhenyu Liang Bowen Zheng Ran Cheng*

Department of Data Science and Artificial Intelligence, The Hong Kong Polytechnic University
Hong Kong SAR, China

*Correspondence: ran-peter.cheng@polyu.edu.hk

ABSTRACT

Complex software systems develop over timescales that exceed the lifespan of any individual coding agent. Most agentic software systems preserve continuity through persistent sessions, memories, managers or shared context. We introduce **EvoX Genesis**¹ (hereafter, **Genesis**), which instead makes the software project persistent while allowing local agents to remain finite-lived. **Genesis** represents software as a *persistent recursive world*: each local world is situated by an accepted version and a repository path, finite-lived agents propose local changes, recursive delegation moves work across paths, and only accepted consequences advance the persistent version history. We evaluate this organization across formation, continuation and redevelopment. Starting from a repository with no compiler implementation, **Genesis** used DeepSeek V4 Flash to build a Rust-based C compiler with about **250k tracked lines**; the run lasted over **120 hours**, archived over 1,000 agent episodes and incurred only **US\$44** in model-token charges. The compiler passed the complete c-testsuite and most LLVM and Csmith tests. In a separate compiler world generated with GLM 5.2, development continued after repeated agent replacement while retaining full test performance. **Genesis** also reimplemented 13 MESA modules with over **100k Fortran lines** as a Rust workspace with nearly **90k Rust lines**; across six numerical workloads, it achieved median speedups of **1.55–6.87×**. These results show that long-horizon software development can be organized around a persistent project rather than a persistent agent.

1 INTRODUCTION

Software tasks end; software systems do not. A repository that survives for months or years accumulates interfaces, tests, architectural commitments, partial solutions and failures that constrain what later contributors can do. Long-horizon software development is therefore not simply a longer coding task. It is a continuing process in which many bounded contributions must remain coherent even when the contributor changes.

Large language models (LLMs) have made repository-level software development increasingly autonomous. Modern coding agents can inspect code, execute tools, modify files and validate their own changes, and benchmarks now extend from issue resolution to multi-step repository construction and software evolution (Ding et al., 2025; Jimenez et al., 2024; Thai et al., 2025; X. Wang et al., 2025; X. Xu et al., 2026; J. Yang et al., 2024). Yet long-horizon systems still face a continuity problem. A larger context window, persistent memory, a manager agent or a shared scratchpad can keep more information available, but these approaches usually preserve continuity by extending some part of the agent process itself. This raises a more basic question: *what must persist when the active agent does not?*

¹Project Website: <https://genesis.evox.group/>

We study an alternative organization in which continuity belongs to the software project. We introduce **EvoX Genesis** (hereafter, **Genesis**), a system that represents software as a *persistent recursive world*. The accepted project state and its history persist; local agents do not. A finite-lived agent enters the project from an accepted version and a repository path, performs a bounded task, proposes a change and terminates. Recursive delegation moves work to more specific paths without immediately changing the accepted version, while validation-gated acceptance determines which consequences become part of the history inherited by later agents.

This formulation separates two timescales that are often conflated: the lifetime of an individual coding agent and the lifetime of the software development process. The former can remain bounded while the latter extends across many episodes, contributors and even foundation models. In this sense, **Genesis** does not attempt to make one agent persistent. It makes the project persist and repeatedly re-instantiates agency within it.

We evaluate this idea through three stages of software development. *Formation* asks whether many finite-lived episodes can accumulate into a complex system from an implementation-empty repository. *Continuation* asks whether an already developed world remains workable after repeated agent replacement and a change of foundation model. *Redevelopment* asks whether the same organization can transform an existing scientific codebase while preserving tested numerical behaviour.

Our contributions are fourfold:

- We formulate a **persistent recursive world** using a minimal version–path model that distinguishes local agency, recursive delegation and accepted software events.
- We implement this formulation in **EvoX Genesis**, where finite-lived manager and executor agents work in path-scoped contexts and only accepted consequences advance the persistent project history.
- We demonstrate large-scale greenfield formation by constructing a C compiler from a repository with no compiler implementation, yielding a 248,989-line repository and broad external test coverage.
- We show that the same project-centered organization supports continuation across foundation-model replacement and scoped redevelopment of MESA modules from Fortran to Rust while preserving the audited numerical behaviour.

2 RELATED WORK

2.1 CODING AGENTS AND LONG-HORIZON SOFTWARE DEVELOPMENT

Repository-level coding agents combine LLMs with file inspection, command execution, editing and test feedback. SWE-bench, SWE-agent and OpenHands established issue-level evaluation and general software-agent platforms (Jimenez et al., 2024; X. Wang et al., 2025; J. Yang et al., 2024). More recent benchmarks extend the horizon to releases, upgrades and sequences of changes (Shastry et al., 2026; Thai et al., 2025; X. Xu et al., 2026), while greenfield benchmarks study repository construction from natural-language specifications (Ding et al., 2025). These settings expose a challenge beyond solving one patch at a time: later work can become harder as interfaces shift, assumptions diverge and technical debt accumulates (Orlanski et al., 2026). **Genesis** focuses on how one accepted project remains developable across many such episodes.

2.2 MEMORY, COORDINATION AND PERSISTENT PROJECT STATE

Agent systems preserve continuity in several ways. Reflection and memory mechanisms retain selected experience, reusable skill libraries preserve procedures, and multi-agent systems maintain coordination through roles and workflows (S. Gao et al., 2026; S. Hong et al., 2024; Shinn et al., 2023; G. Wang et al., 2024). Other approaches store project commitments outside the active conversation or place persistent guidance beside the repository (Gloaguen et al., 2026; Yan et al., 2026). Execution-state systems explicitly record what an agent has observed, changed and attempted (Z. Wang et al., 2026). **Genesis** does not claim that memory, short-lived workers or hierarchical task decomposition are

individually new. Its organizing choice is to treat the accepted project state and history as the object that persists, while agent execution state remains episode-bounded.

Version history is particularly relevant because software already carries accepted change over time. EvoGit allows independent agents to modify and recombine code versions through a Git graph without centralized coordination, explicit message passing or shared memory (Huang et al., 2025). Classical software-engineering work likewise emphasizes that modular decomposition constrains later change (Parnas, 1972) and that change is intrinsic to long-lived software (Lehman, 1980). `Genesis` builds on this view but couples accepted version history to recursive, path-situated agent instantiation and parent-mediated acceptance.

2.3 PROGRAM EVOLUTION AND SCIENTIFIC SOFTWARE REDEVELOPMENT

Repeated generation and evaluation can also organize substantial code change. FunSearch evolves programs against explicit evaluators, AlphaEvolve extends evaluator-guided evolution to more complex algorithmic code, and ERA searches empirical scientific software against explicit quality metrics (Aygün et al., 2026; Novikov et al., 2025; Romera-Paredes et al., 2024). These systems establish that model-generated program variants can be improved through repeated evaluation. Our focus differs: we follow the continuing history of one software project rather than a population or search tree of candidate programs.

Scientific software provides a demanding redevelopment setting because reproducibility depends not only on whether new code runs, but on whether behaviour relevant to scientific use survives environmental and implementation change. Research code can be difficult to reproduce outside its original environment (Moreau et al., 2023; Trisovic et al., 2022); explicit versions and reusable software are therefore central to research-software stewardship (Barker et al., 2022). Recent commentary has also warned that weakly verified AI-assisted changes can threaten scientific-software quality (O’Brien, 2025). We use Modules for Experiments in Stellar Astrophysics (MESA) (Paxton et al., 2011) redevelopment as a first test of whether substantial implementation change can preserve audited numerical behaviour.

3 EVOX GENESIS: PERSISTENT RECURSIVE WORLDS

3.1 PERSISTENT RECURSIVE WORLDS

`Genesis` organizes development around a persistent accepted project rather than a persistent agent identity. At any point, an agent is situated by two coordinates: an accepted version that determines the project it inherits and a repository-relative path that determines where its local responsibility begins. We call this pair a local software world,

$$w = (v, p), \quad (1)$$

where v denotes an accepted software version and p a repository-relative path. The version fixes the complete accepted project state and its inheritable history; the path situates agency within that project. An agent may inspect the complete project represented by v , but it begins from p and receives the context, responsibility and modification scope associated with that location. The path is therefore not a partial copy of the repository and not an additional software state. In the implementation, the accepted project can include source files, path-specific context, constraints, validation results, reusable skills and provenance records.

3.2 TRANSIENT AGENCY AND RECURSIVE DELEGATION

A finite-lived agent A_i receives an episode objective g_i in local world (v, p) and produces a candidate change

$$\Delta_i = A_i((v, p), g_i). \quad (2)$$

The agent can execute multiple model–tool turns during one supervised episode, but its private conversation and scratch state are not intentionally carried as the identity of a later agent. Later work is re-instantiated from an accepted version and path.

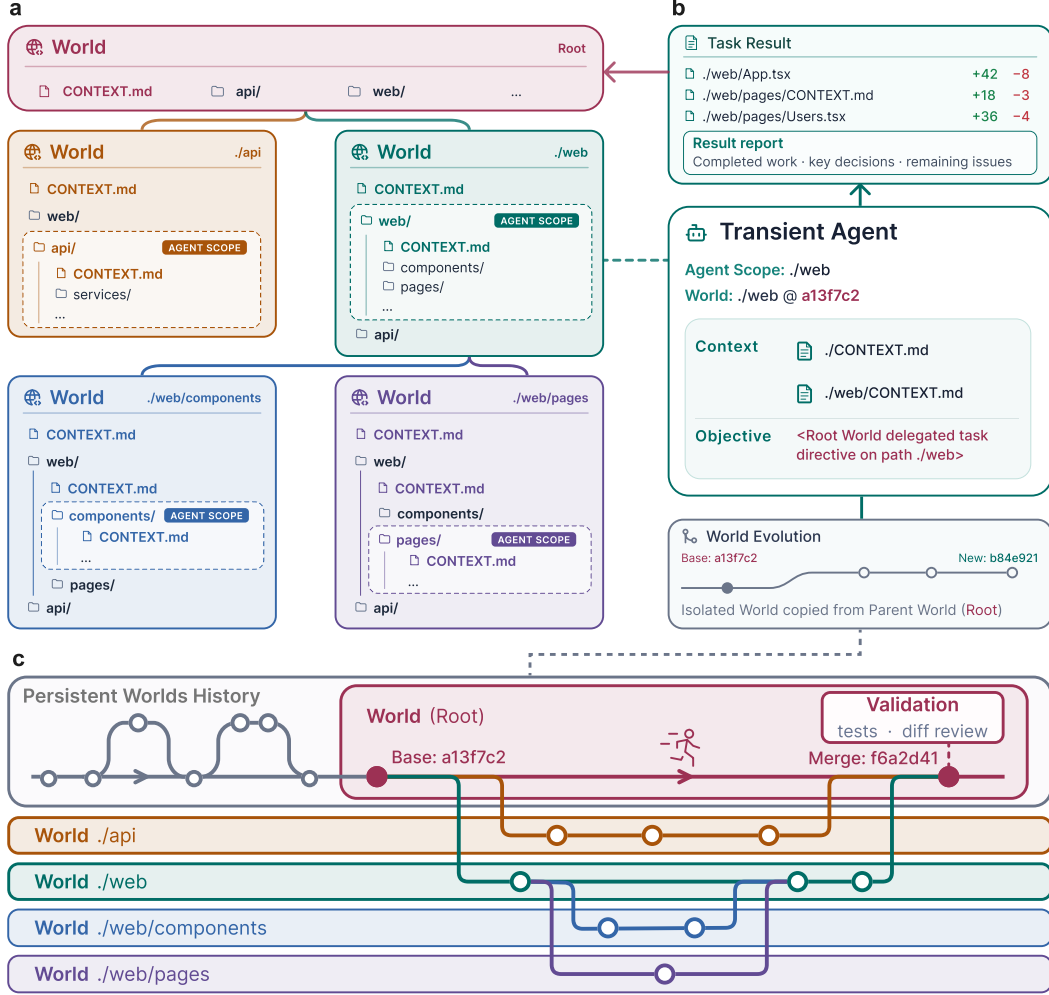


Figure 1. Persistent recursive worlds. **a**, An accepted software version can be viewed from different repository-relative paths, defining local worlds $w = (v, p)$. The version v fixes the accepted project state and history, while the path p sets where an agent starts and what it is responsible for. Recursive delegation $(v, p) \rightsquigarrow (v, q)$ starts a child agent at path q without changing the accepted version v . **b**, A finite-lived agent receives a local objective and proposes a change. The responsible parent accepts, rejects or requests more work using tests, constraints and integration evidence. The isolated worktree is an execution workspace created from the accepted version; it is not a second software world or a partial repository in the formal model. **c**, Only an accepted change creates a software event $(v, p) \rightarrow (v', p')$ and advances the accepted version history. A rejected candidate leaves the accepted version unchanged, and the agent's private execution state ends with the episode.

Recursive delegation changes where work is instantiated without immediately changing the accepted version. A parent agent at path p can create a child at path q in the same version,

$$(v, p) \rightsquigarrow (v, q). \quad (3)$$

The child works from q while v remains fixed, and may recursively delegate again. Leaf executors directly modify the software; root and intermediate managers decompose objectives, delegate subtasks and review returned results. Thus recursion localizes work within the current accepted project without itself advancing the project history.

3.3 VALIDATION AND PERSISTENT LINEAGE

A candidate change becomes persistent only through an accepted software event,

$$(v, p) \rightarrow (v', p'), \quad (4)$$

which advances the accepted project from v to v' . Usually $p' = p$; an accepted rename, move or deletion may instead map the path to p' . The responsible parent decides whether a returned contribution is accepted, rejected or requires further work using the available tests, constraints and integration evidence. A rejected code change leaves the accepted version unchanged. If useful failure information is explicitly stored in context, tests, constraints or provenance, that stored record can become part of a later accepted version even though the rejected code itself does not.

The persistence claim therefore concerns project-specific state rather than every process involved in execution. In the released implementation, directory-scoped nodes assemble context from version-controlled `CONTEXT.md` records, accepted changes are stored through Git commits and protected archive references, and proposed changes are isolated on agent-specific branches and worktrees. Scheduler state, supervised BEAM processes and temporary worktrees provide execution infrastructure rather than the persistent identity of the project. The experiments below evaluate the capabilities of this organization; they do not audit the erasure of every possible provider-side state or isolate the causal contribution of each persistent record.

4 EVALUATION

4.1 EVALUATION DESIGN

We evaluate *Genesis* along three increasingly demanding forms of continuity. *Formation* asks whether bounded local episodes can accumulate into a complex software system from an implementation-empty repository. *Continuity* asks whether an already developed world remains workable after repeated agent replacement and a change of foundation model. *Redevelopment* asks whether an existing scientific codebase can be transformed while preserving tested numerical behaviour. Each setting uses the same project-centered organization but a different starting condition and validation target.

Humans provide the initial task specification, available tools and controller limits. The compiler task includes substantial behavioural and architectural constraints but no compiler implementation or concrete repository decomposition; the continuation study starts from a completed compiler and a high-level continuation objective; the MESA study supplies the reference software, redevelopment objective, compatibility and validation requirements, and performance goals. The archives record objectives, accepted repository histories, agent records and resource summaries, but not a complete audit log of every human action. We therefore report the observed evidence without treating missing intervention records as proof of zero human involvement.

An archived episode is one recorded finite-lived task episode; spawned-agent counts can be larger when some spawned agents are absent from the archive. Retention means that a contribution's commit lies in the ancestry of the final accepted repository, not that the contribution was independently correct. Wall time is measured from the top-level start to finish and differs from summed agent-hours when episodes overlap. Reported US-dollar amounts are foundation-model token charges only and exclude local hardware, storage, controller overhead, networking and labour. Repository line counts are physical-line counts under experiment-specific rules and describe repository size rather than software complexity or feature completeness. The evidence package contains one DeepSeek compiler-formation run, one GLM continuation, one DeepSeek continuation and one MESA-to-Rust redevelopment run; the study therefore describes capabilities under the recorded settings rather than estimating run-to-run success rates. Full protocol, archive and measurement details are provided in the Supplementary Information.

4.2 FORMATION: A C COMPILER FROM SCRATCH

Setup. The first test asks whether a persistent software world can grow into a complex system when there is no implementation to inherit. The run used DeepSeek V4 Flash with `xhigh` reasoning effort and a 150,000-token context-compression threshold. The first root session began from commit `41e087ce90f3`, containing only `.gitignore` and `genesis.toml`; a second root phase inherited the generated repository and a handoff summary. The task requested a clean-room C compiler in Rust for LLVM-centric workflows, including a Clang-compatible command-line interface, standard object-file and linker integration, LLVM IR export, C11 as the primary language target, and mandatory x86 and x86-64 back ends. Direct use or translation of Clang/LLVM source was prohibited. External evaluation sources included c-testsuite, LLVM test programs, LZ4 and SQLite, with Csmith programs

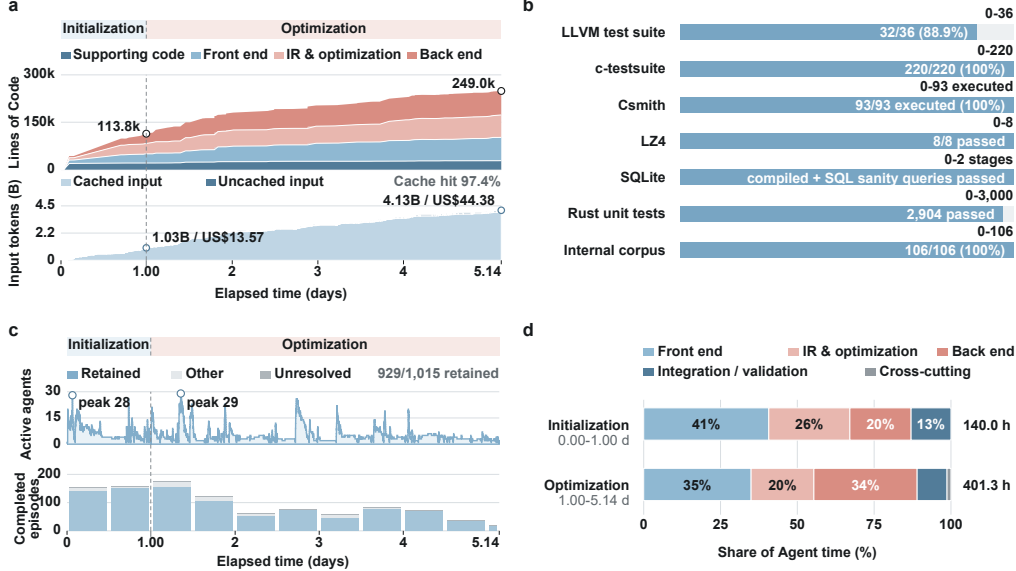


Figure 2. Formation of a C compiler with DeepSeek V4 Flash through recursive development. **a**, Growth of the jcc codebase and cumulative input-token use against elapsed time; the phase boundary separates initialization from optimization. **b**, Final validation across LLVM, c-testsuite, Csmith, LZ4, SQLite, Rust workspace tests and the internal compiler corpus. **c**, Active agents and completed episodes throughout development, with episodes distinguished by whether their contributions were retained in the final accepted repository history. **d**, Agent time partitioned among front-end, intermediate-representation and optimization, back-end, integration and validation, and cross-cutting work during the two phases. Dollar annotations denote model-token charges only, not total compute, infrastructure or labour cost.

generated and executed by the committed harness. The task supplied no compiler implementation code or concrete repository decomposition.

Results. Formation proceeded through recursive accumulation rather than one monolithic generation step. Managers divided the root objective across repository paths, finite-lived agents worked on local pieces, and parent agents reviewed returned changes before those changes entered the accepted project. Over 123.4 h, the run archived **1,019 agent episodes** and reached delegation depth five. The final repository contained **248,989 physical lines** in 750 tracked text files, at a provider-recorded model-token cost of US\$44.38. These counts include comments and blank lines and therefore describe repository size rather than software complexity.

The resulting compiler passed 220/220 reported c-testsuite cases, 32/36 evaluated LLVM cases and 93/93 executed Csmith programs, together with the recorded LZ4 and SQLite checks and 2,904 Rust workspace tests. Because these evaluations have different meanings and denominators, we report them separately rather than combine them into one score.

Interpretation. Compiler construction couples decisions made at different times: choices in the front end constrain type checking, the intermediate representation constrains optimization and code generation, and later integration can expose problems in components that appeared locally complete. No single agent episode spanned this development. Earlier accepted changes instead became the starting point for later agents, while later failures were repaired with the rest of the project in place. The experiment therefore establishes large-scale formation under the stated task contract and makes the accumulation of bounded contributions into one interdependent software system directly observable.

4.3 CONTINUITY: DEVELOPMENT ACROSS FOUNDATION-MODEL REPLACEMENT

Setup. The second test asks whether a completed software world can continue after the agents that built it are gone, and whether continuation remains possible when the foundation model itself

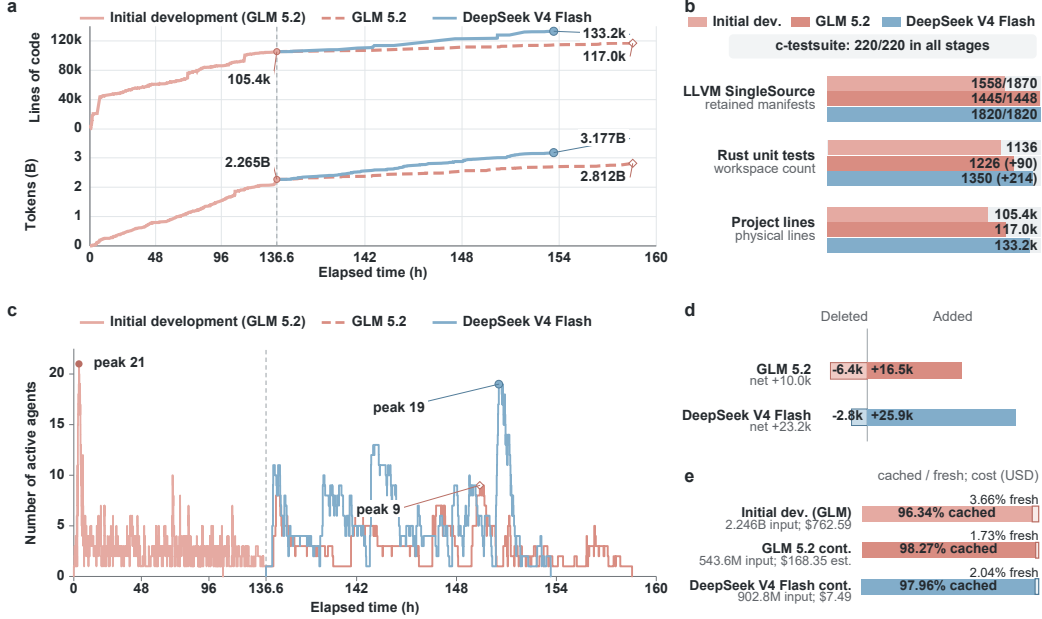


Figure 3. Continuation of the same compiler world with GLM 5.2 and DeepSeek V4 Flash. **a**, Codebase growth and cumulative token use across initial GLM 5.2 development and the two continuations from the same completed GLM jcc world. **b**, Compiler validation and project growth for the three recorded snapshots. The retained LLVM SingleSource test sets differ among snapshots, so the fractions are reported within each snapshot rather than compared on one fixed test set. **c**, Concurrent agent activity across initial development and both continuation runs. **d**, Lines added and deleted during each continuation relative to the shared starting codebase. **e**, Cached and fresh input-token fractions and corresponding recorded or reconstructed model-token costs. The costs were obtained differently for the two continuation runs, and the runs did not use matched resources, so no normalized dollar-efficiency comparison is made.

changes. This study uses a compiler history separate from the DeepSeek formation run. Both continuation branches start from the same completed jcc repository generated with GLM 5.2 at commit 37216cfa254a, receive the same completed project, prior-task context, user-level objective, evaluation families and controller limits, and run on the same dedicated machine. One branch continues with GLM 5.2 and the other switches to DeepSeek V4 Flash. Both use maximum depth eight, retry limit 15, 2,048 root turns, 128 delegated turns and a 150,000-token compression threshold. The retained LLVM test sets differ across snapshots, and the two branches do not use a matched token or wall-clock budget; the experiment is therefore descriptive rather than a controlled model comparison.

Results. Both branches resumed development through repeated replacement of finite-lived agents. GLM 5.2 passed 1,445/1,448 cases in its retained LLVM test set, whereas DeepSeek V4 Flash passed 1,820/1,820. GLM used 98 agents and reached observed depth four, while DeepSeek used 178 agents and reached depth eight. The retained LLVM test sets were not identical, so these pass rates describe each completed snapshot rather than a head-to-head comparison on one fixed test set. Even so, both branches advanced the same inherited compiler instead of rebuilding it from the beginning.

Interpretation. Model replacement changes the process that proposes local changes, yet both branches resumed from the same completed compiler and then diverged in agent counts, delegation depth, commit history, code churn and final repository size. The inherited world therefore acted as a common starting point and a set of obligations rather than a script that fixed the next state. Persistence constrained what had to be inherited without fixing how the future had to unfold. The significance of this experiment is continuation across model replacement, not a ranking of GLM and DeepSeek.

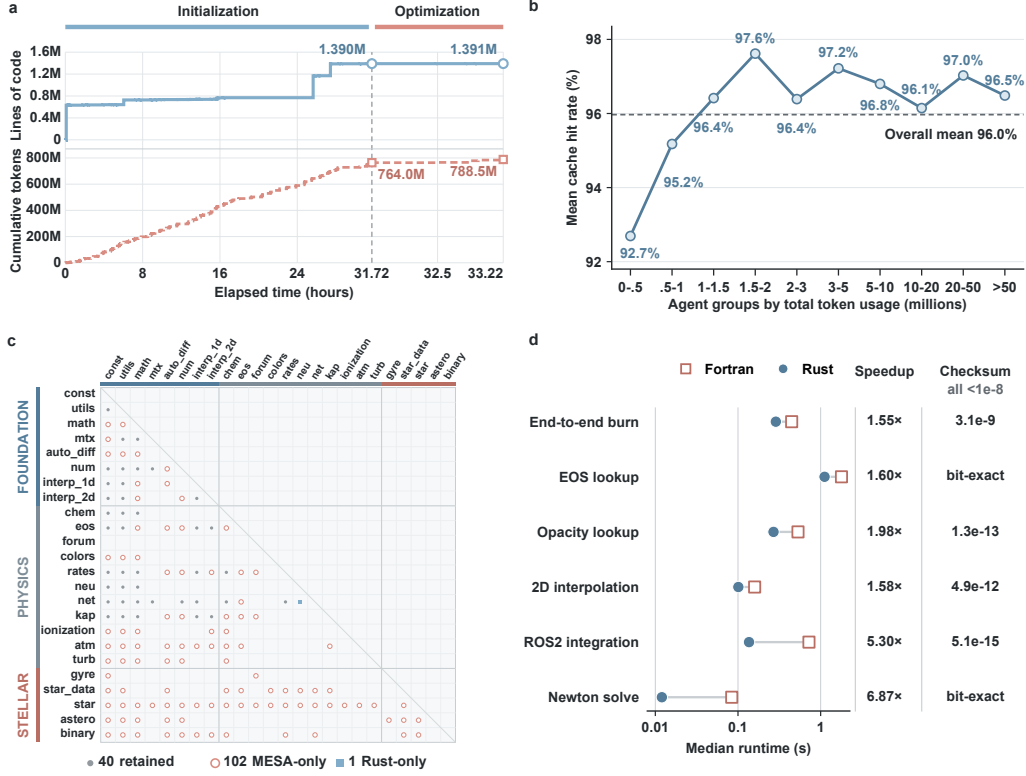


Figure 4. Redevelopment of selected MESA modules in Rust. **a**, Repository growth and cumulative input-token use during the MESA-to-Rust migration. The plotted repository line count is the size of the generated repository, not the number of MESA source lines migrated. **b**, Mean prompt-cache hit rates for agent groups defined by total token use. **c**, Dependencies among the mapped modules, classified as present in both dependency graphs, absent as direct Rust crate dependencies, or newly present in Rust. **d**, Numerical agreement and median runtime for corresponding Fortran and Rust implementations across six audited workloads.

4.4 REDEVELOPMENT: MESA FROM FORTRAN TO RUST

Setup. The third test asks whether Genesis can inherit scientific software, change its implementation language and preserve the numerical behaviour that matters. Modules for Experiments in Stellar Astrophysics (MESA)² is an open-source suite for one-dimensional stellar-evolution calculations (Paxton et al., 2011). The study used a lightly modified MESA fork at commit 461dcba94f33 as a read-only reference and DeepSeek V4 Flash to reimplement 13 mapped module directories as corresponding Rust crates. The reported scope covers basic numerical and physics modules and contains **139,414 physical Fortran lines** including module-level tests; higher-level MESA engines such as `star`, `astero` and `binary` are outside the migration. Numerical validation used six standalone workloads covering end-to-end burn, EOS lookup, opacity lookup, two-dimensional interpolation, ROS2 integration and Newton solve. Each reported runtime is the median of 25 post-warm-up measurements per implementation; full build flags, CPU pinning and the separate 40-run burn check are reported in the Supplementary Information.

Results. The run completed the scoped rewrite in 33.22 h, spawned 272 agents and produced a Rust workspace containing **89,946 physical Rust lines** including tests and benches. The workspace passed 1,052 tests with no failures and 18 ignored tests, at a provider-recorded model-token cost of US\$10.64. Across the six audited numerical workloads, EOS lookup and Newton solve were bit-exact; relative checksum differences for the other four workloads ranged from 5.1×10^{-15} to 3.1×10^{-9} . Rust had

²MESA source repository: <https://github.com/MESAHub/mesa/>

the lower median runtime in all six workloads, with measured speedups from **1.55×** to **6.87×**. The timing comparison is specific to the reported builds, host and benchmark harness.

Interpretation. The compiler experiment began with no implementation, whereas the MESA experiment began with a mature system whose numerical behaviour already had meaning. The task was therefore not simply to produce Rust code: implementation could change, but the tested numerical relationships carried by the original software had to survive. The resulting workspace differs substantially from the Fortran source in size and dependency structure, yet the audited workloads remained numerically aligned. This moves Genesis from constructing a new software world to redeveloping an existing one while preserving the tested behaviour that gave the original code its scientific value.

5 DISCUSSION

5.1 PROJECT-CENTERED CONTINUITY

The central design choice in Genesis is where continuity resides. Many long-horizon agent systems extend an agent process through longer context, explicit memory, a persistent manager or shared state. Genesis instead makes the accepted project the persistent object. Code, path-specific context, constraints, validation results and history remain available to later work, while local agents can terminate. The software world is therefore not an auxiliary memory attached to a long-lived agent; it is the project state from which successive agents are instantiated.

This interpretation does not make the software world itself an agent. The world has no persistent private intention, conversation or cognitive identity. Reasoning remains in finite-lived agents. The accepted version specifies what exists, the path situates local responsibility, and parent-mediated acceptance determines which consequences become part of the history inherited by later work. Persistence and agency are therefore separated rather than transferred from one subject to another.

The three experiments cover formation, continuity and redevelopment. In the compiler-formation run, more than a thousand finite-lived episodes accumulated into one interdependent software system. In the continuation study, a completed compiler remained developable after repeated agent replacement and a foundation-model change. In the MESA study, the same project-centered organization supported substantial scientific-software redevelopment while preserving the audited numerical behaviour. Together, these observations show that, under the reported settings, the lifetime of the development process can exceed the lifetime of the agents acting within it.

The continuation experiment further shows that persistence does not prescribe a single future. Starting from the same completed compiler, GLM 5.2 and DeepSeek V4 Flash produced different delegation depths, agent counts, commit histories, code churn and final repository sizes while both continued development. The accepted project supplied a shared past and a common set of obligations, but different models took different routes forward. Likewise, the MESA experiment raises a stronger standard for inheritance: a scientific successor is useful only if externally meaningful behaviour survives implementation change.

5.2 EVIDENCE AND CAUSAL BOUNDARIES

The present experiments establish capability across three distinct regimes, but they do not yet provide a complete causal decomposition of the mechanism. Recursion was operationally central in all reported runs, reaching observed depths of five in compiler formation, four and eight in the two continuation branches, and four in MESA redevelopment. These observations show that recursive delegation was extensively used, but they do not establish that recursion is causally superior to flat or alternative organizations. Similarly, the continuation results show that development can proceed across agent and foundation-model replacement, but they do not by themselves determine which persistent records are necessary for that continuity.

The clearest next experiments therefore concern mechanism causality. One test holds executable code fixed while changing accepted non-code development records, asking whether future construction changes under the same model, task and budget. A second compares fresh agents with persistent agents while holding the saved project state fixed. Together with flat-organization, hierarchical-acceptance

and validation ablations, these experiments can determine which components of the persistent recursive world are necessary for the observed long-horizon capabilities. Until such controlled comparisons are available, the present results should be read as evidence that the reported organization supports formation, continuity and redevelopment, rather than as proof that every component is individually necessary.

5.3 SCOPE OF SOFTWARE EVOLUTION

Our use of *software evolution* is practical rather than biological. It denotes the trajectory through which a software system forms, inherits earlier structure and changes through accepted version history. Agents do not reproduce, and the present study does not claim Darwinian evolution, open-ended self-modification or learning of the foundation-model parameters. Instead, variation arises from local candidate changes, persistence from accepted project history, and continued development from repeated re-instantiation of finite-lived agency within that history.

Likewise, *autonomous* is bounded rather than absolute. Humans provide the initial objectives, available tools, validation sources and controller limits; within those conditions, *Genesis* decomposes work, instantiates local agents, executes development actions and determines which validated consequences enter the persistent project. The claim is therefore not that software develops without external goals or infrastructure, but that a long-horizon development process can continue without requiring a persistent intelligent agent to carry its identity or history.

6 CONCLUSION

EvoX *Genesis* reorganizes long-horizon software development around a persistent recursive project rather than a persistent agent. A local world is situated by an accepted version and a repository path; finite-lived agents propose changes; recursive delegation moves work across paths; and accepted consequences advance the version history inherited by later agents. Under this organization, the reported runs formed a large C compiler from an implementation-empty repository, continued an existing compiler after foundation-model replacement and redeveloped selected MESA modules in Rust while preserving the audited numerical behaviour. These results motivate a broader hypothesis: persistent project state can carry software development across changing episodes of intelligence. Determining exactly which persistent records and recursive mechanisms are necessary is the next step.

REFERENCES

- Ayguin, E., Belyaeva, A., Comanici, G., Coram, M., Cui, H., et al. (2026). “An AI system to help scientists write expert-level empirical software”. In: *Nature* 654, pp. 909–916. doi: 10.1038/s41586-026-10658-6.
- Barker, M., Hong, N. P. C., Katz, D. S., Lamprecht, A.-L., Martinez-Ortiz, C., et al. (2022). “Introducing the FAIR Principles for research software”. In: *Scientific Data* 9, p. 622. doi: 10.1038/s41597-022-01710-x.
- Ding, J., Long, S., Pu, C., et al. (2025). “NL2Repo-Bench: Towards Long-Horizon Repository Generation Evaluation of Coding Agents”. In: *arXiv preprint arXiv:2512.12730*. doi: 10.48550/arXiv.2512.12730. arXiv: 2512.12730 [cs.CL].
- Gao, S., Zeng, W., Yu, Z., Wangni, J., Wang, C., Cai, K., He, S., and Lyu, M. R. (2026). “SWE-MeM: Learning Adaptive Memory Management for Long-Horizon Coding Agents”. In: *arXiv preprint arXiv:2606.28434*. doi: 10.48550/arXiv.2606.28434.
- Gloaguen, T., Mündler, N., Müller, M., Raychev, V., and Vechev, M. (2026). “Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents?” In: *arXiv preprint arXiv:2602.11988*. doi: 10.48550/arXiv.2602.11988.
- Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., and Schmidhuber, J. (2024). “MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework”. In: *International Conference on Learning Representations*.
- Huang, B., Cheng, R., and Tan, K. C. (2025). “EvoGit: Decentralized Code Evolution via Git-Based Multi-Agent Collaboration”. In: *arXiv preprint arXiv:2506.02049*. doi: 10.48550/arXiv.2506.02049. arXiv: 2506.02049 [cs.SE].

- Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. (2024). “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?” In: *International Conference on Learning Representations*.
- Lehman, M. M. (1980). “Programs, Life Cycles, and Laws of Software Evolution”. In: *Proceedings of the IEEE* 68.9, pp. 1060–1076. doi: 10.1109/PROC.1980.11805.
- Moreau, D., Wiebels, K., and Boettiger, C. (2023). “Containers for computational reproducibility”. In: *Nature Reviews Methods Primers* 3, p. 50. doi: 10.1038/s43586-023-00236-9.
- Novikov, A., Vü, N., Eisenberger, M., et al. (2025). “AlphaEvolve: A coding agent for scientific and algorithmic discovery”. In: *arXiv preprint arXiv:2506.13131*. doi: 10.48550/arXiv.2506.13131. arXiv: 2506.13131 [cs.AI].
- O’Brien, G. (2025). “Threats to scientific software from over-reliance on AI code assistants”. In: *Nature Computational Science* 5, pp. 701–703. doi: 10.1038/s43588-025-00845-2.
- Orlanski, G., Roy, D., Yun, A., Shin, C., Gu, A., Ge, A., Adila, D., Roberts, N., Sala, F., and Albarghouthi, A. (2026). “SlopCodeBench: Benchmarking How Coding Agents Degrade Over Long-Horizon Iterative Tasks”. In: *arXiv preprint arXiv:2603.24755*. Version 2. doi: 10.48550/arXiv.2603.24755.
- Parnas, D. L. (1972). “On the criteria to be used in decomposing systems into modules”. In: *Communications of the ACM* 15.12, pp. 1053–1058. doi: 10.1145/361598.361623.
- Paxton, B., Bildsten, L., Dotter, A., Herwig, F., Lesaffre, P., and Timmes, F. (2011). “Modules for Experiments in Stellar Astrophysics (MESA)”. In: *The Astrophysical Journal Supplement Series* 192.1, p. 3. doi: 10.1088/0067-0049/192/1/3. arXiv: 1009.1622 [astro-ph.SR].
- Romera-Paredes, B., Barekatin, M., Novikov, A., et al. (2024). “Mathematical discoveries from program search with large language models”. In: *Nature* 625, pp. 468–475. doi: 10.1038/s41586-023-06924-6.
- Shastri, K. N. A., Senrayan, G., Satapara, S., Panda, P., and Devaguptapu, C. (2026). “Beyond Isolated Tasks: A Framework for Evaluating Coding Agents on Sequential Software Evolution”. In: *arXiv preprint arXiv:2604.03035*. doi: 10.48550/arXiv.2604.03035.
- Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. (2023). “Reflexion: Language Agents with Verbal Reinforcement Learning”. In: *Advances in Neural Information Processing Systems*. Vol. 36.
- Thai, M. V. T., Le, T., Manh, D. N., Nhat, H. P., and Bui, N. D. Q. (2025). “SWE-EVO: Benchmarking Coding Agents in Long-Horizon Software Evolution Scenarios”. In: *arXiv preprint arXiv:2512.18470*. doi: 10.48550/arXiv.2512.18470.
- Trisovic, A., Lau, M. K., Pasquier, T., and Crosas, M. (2022). “A large-scale study on research code quality and execution”. In: *Scientific Data* 9, p. 60. doi: 10.1038/s41597-022-01143-6.
- Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. (2024). “Voyager: An Open-Ended Embodied Agent with Large Language Models”. In: *Transactions on Machine Learning Research*. URL: <https://openreview.net/forum?id=ehfRiF0R3a>.
- Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., et al. (2025). “OpenHands: An Open Platform for AI Software Developers as Generalist Agents”. In: *International Conference on Learning Representations*.
- Wang, Z., Xu, Y., Li, C., Peng, C., Adams, B., Hassan, A. E., and Chen, T.-H. (2026). “Turning Interaction History into Execution State: A Runtime Layer for Long-Horizon Coding Agents”. In: *arXiv preprint arXiv:2608.00808*. doi: 10.48550/arXiv.2608.00808. arXiv: 2608.00808 [cs.SE].
- Xu, X., Yang, R., Shen, H., Xu, W., Gao, B., Wu, R., et al. (2026). “RoadmapBench: Evaluating Long-Horizon Agentic Software Development Across Version Upgrades”. In: *arXiv preprint arXiv:2605.15846*. doi: 10.48550/arXiv.2605.15846.
- Yan, L., Chen, X., and Zhang, X. (2026). “When the Specification Emerges: Benchmarking Faithfulness Loss in Long-Horizon Coding Agents”. In: *arXiv preprint arXiv:2603.17104*. doi: 10.48550/arXiv.2603.17104.
- Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. (2024). “SWE-agent: Agent–Computer Interfaces Enable Automated Software Engineering”. In: *Advances in Neural Information Processing Systems*. Vol. 37, pp. 50528–50652. doi: 10.52202/079017-1601.

PERSISTENT RECURSIVE WORLDS ENABLE AUTONOMOUS SOFTWARE EVOLUTION

Beichen Huang Zhenyu Liang Bowen Zheng Ran Cheng*

Department of Data Science and Artificial Intelligence, The Hong Kong Polytechnic University
Hong Kong SAR, China

*Correspondence: ran-peter.cheng@polyu.edu.hk

CONTENTS

1	Introduction	1
2	Related Work	2
2.1	Coding agents and long-horizon software development	2
2.2	Memory, coordination and persistent project state	2
2.3	Program evolution and scientific software redevelopment	3
3	EvoX Genesis: Persistent Recursive Worlds	3
3.1	Persistent recursive worlds	3
3.2	Transient agency and recursive delegation	3
3.3	Validation and persistent lineage	4
4	Evaluation	5
4.1	Evaluation design	5
4.2	Formation: a C compiler from scratch	5
4.3	Continuity: development across foundation-model replacement	6
4.4	Redevelopment: MESA from Fortran to Rust	8
5	Discussion	9
5.1	Project-centered continuity	9
5.2	Evidence and causal boundaries	9
5.3	Scope of software evolution	10
6	Conclusion	10
7	Formal Definition of Persistent Recursive Worlds	14
7.1	Minimal model	14
7.2	Local software worlds	14
7.3	Finite-lived agents	14
7.4	Accepted changes	14

7.5	Recursive delegation	14
7.6	Persistent recursive worlds	15
7.7	Reopening and structural changes	15
7.8	Scope of the model	15
8	Evidence and Measurement Conventions	15
8.1	Evidence sources	15
8.2	Repository versions	15
8.3	Measurement and accounting conventions	16
9	Formation: A C Compiler from Scratch	16
9.1	Evidence source	16
9.2	Task and initial specification	16
9.3	Initial repository and run phases	16
9.4	Human-provided information	16
9.5	Evaluation targets	16
9.6	External validation sources	17
9.7	Run configuration and archive structure	17
9.8	Developmental lineage and complete results	17
9.8.1	Resource use and repository size	18
9.9	Archive completeness	20
10	Continuity: Development Across Foundation-Model Replacement	20
10.1	Shared starting world	20
10.2	Evaluation protocol	21
10.3	Run settings, lineage and results	21
10.3.1	Resources, repository growth and validation	23
11	Redevelopment: MESA from Fortran to Rust	25
11.1	Starting system and run settings	25
11.2	Migration scope and module mapping	25
11.3	Run timeline and concurrency	27
11.4	Numerical validation and runtime	28
11.5	Dependency structure and recursive context	29
12	Evidence Boundaries and Audit Notes	31
12.1	What each experiment establishes	31
12.2	Observed failures and missing records	31
12.3	Causal tests not performed in this study	32

7 FORMAL DEFINITION OF PERSISTENT RECURSIVE WORLDS

7.1 MINIMAL MODEL

The minimal model uses four objects: an accepted software version v , a repository-relative path p , a finite-lived agent computation A_i and an accepted software event. Recursive organization, Context, constraints, validation results, skills and provenance are stored in the accepted version rather than modeled as separate state variables.

7.2 LOCAL SOFTWARE WORLDS

A local software world is

$$w = (v, p). \quad (5)$$

The accepted version v fixes the complete project and its saved history. The path p says where the agent starts and which Context, responsibility and scope apply. An agent may inspect the complete project represented by v ; p is neither a partial copy of the repository nor a separate software state.

A version is an accepted software state, not necessarily a bare Git SHA. In the runtime it can be a validated commit, checkpoint or merge together with the Context, constraints, tests, skills and provenance that later work can inherit. Intermediate worktrees and rejected candidates are not accepted versions.

7.3 FINITE-LIVED AGENTS

For an episode-level objective g_i in local world (v, p) , a finite-lived agent produces a candidate change

$$\Delta_i = A_i((v, p), g_i). \quad (6)$$

An agent may keep private execution state during its episode. The model uses two roles: managers at root and intermediate nodes decompose objectives, delegate work and judge returned results; leaf executors directly modify the software. Runtime labels such as codebase lead, codebase investigator and task scheduler are implementation labels, not additional roles in the model. Private execution state is not part of the accepted version history.

7.4 ACCEPTED CHANGES

A candidate enters the accepted version history only through an accepted event

$$(v, p) \longrightarrow (v', p'). \quad (7)$$

The event advances the accepted project from v to v' . Usually $p' = p$; a rename, move or deletion can instead require an explicit path mapping. For delegated work, the parent decides whether to accept, reject or request further work using the relevant scope checks, tests, integration results and, where needed, scientific checks.

A rejected candidate does not advance the accepted version. If a failure reason is saved in Context, tests, constraints or provenance, that saved record is a separate accepted event; the rejected code itself is not.

7.5 RECURSIVE DELEGATION

Recursive delegation changes where the agent works without changing the accepted version:

$$(v, p) \rightsquigarrow (v, q). \quad (8)$$

A parent agent working from p invokes a child at q in the same accepted version v . This does not create a new accepted version or automatically accept the child's work. Intermediate children may delegate again; leaf executors return direct software changes. The version advances only after the parent accepts a returned result.

The two operations differ as follows:

Operation	Version	Working path
Recursive delegation $(v, p) \rightsquigarrow (v, q)$	unchanged	changes from p to q
Accepted event $(v, p) \rightarrow (v', p')$	advances to v'	remains p or is explicitly mapped

7.6 PERSISTENT RECURSIVE WORLDS

A persistent recursive world combines two kinds of continuity:

1. an *accepted version history*, in which accepted results advance the project from one version to the next; and
2. *recursive work across paths*, in which new finite-lived agents are started from different paths within the current accepted version.

The model does not require private conversation to persist across episodes. Later work starts from an accepted version and path. What persists is the accepted project and its history, together with Context, constraints, validation results, skills and saved failure records.

7.7 REOPENING AND STRUCTURAL CHANGES

The words modify, expand, integrate, close, reopen, prune and regrow describe kinds of accepted changes. They are not additional mathematical objects. Closing or reopening changes the accepted version history; pruning and regrowth can require several accepted events and path mappings; recursive delegation itself does not change the version.

Delegating to a child path does not itself change the software. A proposed structural change becomes part of the world only after its result is accepted into a new version.

7.8 SCOPE OF THE MODEL

The model describes software development carried by accepted versions while individual agent computations are short-lived. It is not a population model of reproducing agents, and it does not say that the foundation model learns new parameters during a run. What accumulates is the accepted project history and the Context and constraints available to later work.

8 EVIDENCE AND MEASUREMENT CONVENTIONS

8.1 EVIDENCE SOURCES

The Supplementary Information uses three evidence packages: compiler formation, compiler continuation and MESA-to-Rust migration. They contain the task instructions, archived run summaries, repository versions and the tables, figures or source data used below.

8.2 REPOSITORY VERSIONS

Git commit IDs identify the reported repository versions. They do not identify non-Git logs or validation reports byte for byte.

Table S1. Repository commits used in the three experiments.

Repository	Version	Commit ID (SHA-1)
jcc	DeepSeek generated jcc	a8c116ec7ed3d59479ad3be8fed3e63384f50a87
jcc	GLM generated jcc	37216cfa254a40e40af48cd528743fd7f2d6737c
jcc	GLM generated jcc continued with DeepSeek	10077f65686102deef2c228e8a49b0569218ec48
jcc	GLM generated jcc continued with GLM	ab58a91460adb24e9349a554e02d4522d3ae2db4
mesa-rs	DeepSeek ported mesa	2ad071e658f8d4de21aa9f0e906592c364e10023

Both continuation runs start from the GLM-generated Task 1 compiler at commit 37216cfa254a. The DeepSeek compiler at commit a8c116ec7ed3 comes from the separate formation study and is not the starting point for continuation.

8.3 MEASUREMENT AND ACCOUNTING CONVENTIONS

Unless a subsection states otherwise, the following conventions are used across the evidence package. An *episode* is one row in `archive_records`. Wall time is the top-level finish time minus the start time. Agent-hours are the sum of archived episode durations, so they exceed wall time when episodes overlap. Parent-child tree statistics exclude records whose `parent_id` is absent from the archive and any descendants of such records. A record is called retained when its commit is an ancestor of the final repository; this shows integration, not correctness. Physical repository lines count every line in tracked text files, including blank lines, comments and non-Rust files. They measure repository size, not compiler implementation size alone. Token and cost quantities are reported from the available run records and are not normalized across experiments when the underlying accounting differs. Validation families with different denominators or meanings are kept separate rather than collapsed into a single score.

9 FORMATION: A C COMPILER FROM SCRATCH

9.1 EVIDENCE SOURCE

The main run record for compiler formation is `data/archive-deepseek-llvm.json`. Repository growth and composition were recomputed from the Git repository at final commit `a8c116ec7ed3d59479ad3be8fed3e63384f50a87`; validation results come from the final root result and the committed test-harness documentation. This DeepSeek run is separate from the older GLM 5.2 compiler used in Section 10.

9.2 TASK AND INITIAL SPECIFICATION

The compiler experiment was one Genesis run from 2 August to 7 August 2026 (UTC) using DeepSeek V4 Flash. The task asked for an independent C compiler written in Rust for LLVM-oriented toolchains and tests. It required a Clang-compatible command-line interface, standard object-file and linker integration, LLVM IR export and evaluation with the LLVM test suite. The compiler used a custom typed CIR with arena/index-based storage rather than LLVM IR, and direct use or translation of Clang/LLVM source was prohibited. C11 was the main language target, C23 was a stretch goal, x86 and x86-64 were required back ends and AArch64 was optional. The task therefore supplied substantial high-level design, toolchain and testing constraints. The run tests implementation from a repository with no compiler code; it does not test architecture-free formation.

9.3 INITIAL REPOSITORY AND RUN PHASES

The first root session began at commit `41e087ce90f3`, which contained only `.gitignore` and `genesis.toml` and no compiler implementation. Each delegated agent worked in its own Git worktree. The second root session began from the first session’s final commit, `04d64b89d504`, and received the accepted compiler repository plus a handoff summary of completed work, validation results, known gaps and next steps. The two root sessions are two phases of the same run, not independent repeats.

9.4 HUMAN-PROVIDED INFORMATION

The root objective specified the implementation language, high-level compiler organization, typed intermediate representation, command-line compatibility, target architectures and test sources. Humans did not provide compiler code or concrete module implementations. The run therefore tests automated implementation under a detailed human specification, not architecture-free formation.

9.5 EVALUATION TARGETS

The language targets were C11 and selected C23 extensions. Compiler behaviour was checked with the recorded test suites and project compilation tests described below. These tests do not prove complete C11 conformance.

9.6 EXTERNAL VALIDATION SOURCES

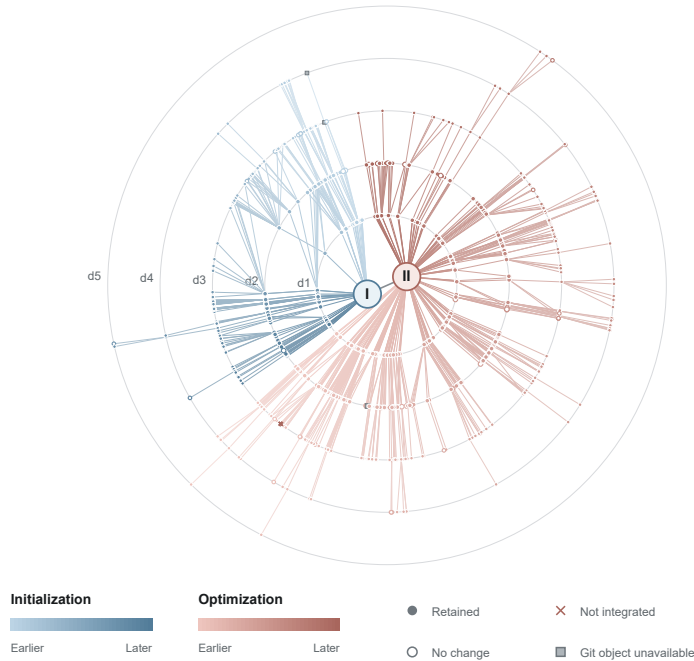
LLVM (upstream: <https://github.com/llvm/llvm-project>) is used as a toolchain and test source, not as a C-language standard. Here, “LLVM-compatible” means that jcc works with LLVM-oriented workflows and is tested on LLVM test programs; it does not mean that jcc reused LLVM code or LLVM’s internal IR. c-testsuite, the LLVM test suite, LZ4 and SQLite supplied external test programs, and Csmith programs were generated by the committed harness. The results are reported separately because these tests measure different things.

9.7 RUN CONFIGURATION AND ARCHIVE STRUCTURE

The run had two root sessions in sequence. The archive contains 1,019 episode records, while the top-level agent_count reports 1,065 spawned agents, so archive-based organization statistics use only the records that are present. Four archived records point to missing parent IDs and have no archived descendants, leaving 1,015 records for the reconstructed parent–child tree. Token and cost totals come from the top-level usage record. Archived objective fields contain task instructions but not complete system prompts, message histories or tool outputs.

9.8 DEVELOPMENTAL LINEAGE AND COMPLETE RESULTS

Run totals, archived episodes, records in the reconstructed parent–child tree, Git history, repository line counts and test results refer to different sets of records, so they are reported separately.



Supplementary Fig. S5. DeepSeek compiler-development tree. The radial tree shows the two root sessions and all 1,015 records used in the reconstructed parent–child tree after four records with missing parents were excluded. Radial distance shows delegation depth. Blue and coral show initialization and optimization; shade shows completion order within each phase. Symbols mark retained, no-change, not-integrated and Git-object-unavailable outcomes.

Table S2. Compiler archive and integration counts.

Quantity	Value
Raw archived records	1,019
Top-level <code>agent_count</code> field	1,065
Direct missing-parent records	4
Excluded descendants	0
Records in reconstructed tree	1,015
Initialization records	312
Optimization records	703
Retained	929
No change	78
Not integrated	5
Git object unavailable	3
First-parent commits	327
Maximum observed delegation depth	5
Peak active episodes	29

Retention means that a contribution remains in the Git history leading to the final repository; it does not mean that the contribution was independently correct. The configured maximum delegation depth was 8, and the deepest archived record was at depth 5.

9.8.1 RESOURCE USE AND REPOSITORY SIZE

The run lasted 123.402 h. Summed episode durations were 666.385 agent-hours because episodes overlapped. Cached input accounted for 97.382% of input tokens.

Table S3. Compiler-formation resource use.

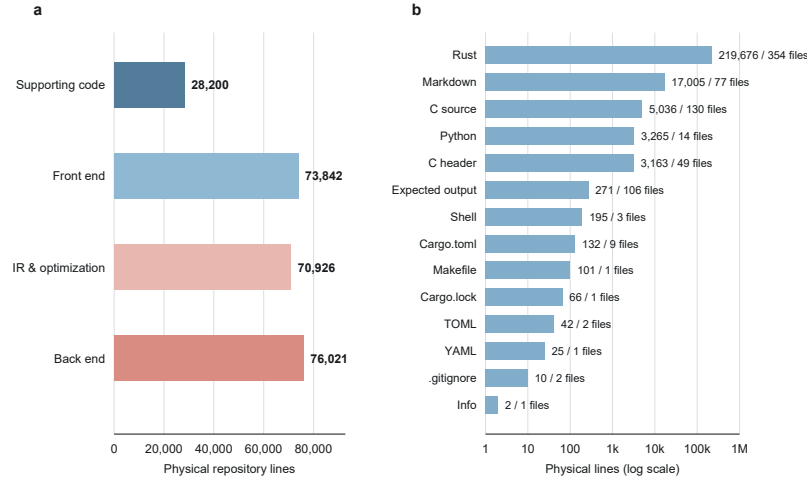
Metric	Observed value
Elapsed wall time	123.402 h
Raw agent-hours	666.385 h
Median episode duration	12.76 min
Input tokens	4,134,593,954
Cached input tokens	4,026,336,896
Uncached input tokens	108,257,058
Cached input/input	97.382%
Output tokens	64,092,688
Total tokens	4,198,686,642
Logged cost	US\$44.3760

Table S4. Compiler-formation resource use by phase.

Phase	Records	Wall time (h)	Agent-hours	Total tokens	Cost (US\$)
Initialization	312	23.905	163.925	1,052,907,912	13.5715
Optimization	707	99.497	502.460	3,145,778,730	30.8045
Total	1,019	123.402	666.385	4,198,686,642	44.3760

Phase totals come from the two non-overlapping root usage records. Per-episode usage records are not summed for billing because parent records include usage from their descendants.

Physical-line counts include blank lines and comments in every tracked text file. The final repository contains 248,989 physical lines across 750 text files and no tracked binary files; 219,676 lines are in 354 Rust source files. The remaining lines include documentation, tests and validation material. The 248,989 total is therefore a repository-size measure, not a count of compiler implementation lines or a measure of software quality.



Supplementary Fig. S6. Final DeepSeek compiler repository. The final tracked repository contains 248,989 physical lines in 750 text files, including 219,676 Rust lines in 354 `.rs` files. **a**, Lines grouped by functional category. **b**, Line and file counts by tracked file type on a logarithmic scale. Counts include comments and blank lines and describe the whole repository, not compiler implementation alone.

Table S5. Final compiler repository lines by tracked text-file type.

File type	Files	Physical lines	Share
Rust	354	219,676	88.23%
Markdown	77	17,005	6.83%
C source	130	5,036	2.02%
Python	14	3,265	1.31%
C header	49	3,163	1.27%
Expected output	106	271	0.11%
Shell	3	195	0.08%
Cargo.toml	9	132	0.05%
Makefile	1	101	0.04%
Cargo.lock	1	66	0.03%
TOML	2	42	0.02%
YAML	1	25	0.01%
.gitignore	2	10	0.00%
Info	1	2	0.00%
Total	750	248,989	100.00%

Table S6. Final compiler test results and limits.

Test	Measure	Result	Rate	Notes
LLVM test suite	Passing cases	32/36	88.9%	Four reported cases did not pass
c-testsuite	Passing tests	220/220	100.0%	Complete reported c-testsuite set
Csmith	Executed random programs	93/93	100.0%	Seven of 100 seeds skipped; zero executed failures
LZ4	Essential checks	8/8	100.0%	All reported essential checks passed
SQLite	Test stages	2/2	100.0%	Compile/link and deterministic SQL checks; not the upstream suite
Rust unit tests	Workspace count	2,904 passed	—	One intentional ignore; no fixed external denominator
Internal corpus	Compiler cases	106/106	100.0%	86 compile-run and 20 compile-fail cases

The test sets contain different numbers and kinds of cases. `SQLite` used deterministic shell queries rather than the full upstream suite; `Csmith` reports 93 executed seeds, with seven skipped. These

results show that the run produced a working C compiler under the stated task specification. They do not establish full C11 conformance, production readiness or a run-to-run success rate.

Table S7. Compiler task and run settings.

Item	Setting
Run setup	Genesis; two sequential root sessions; per-agent Git worktrees
Model	DeepSeek V4 Flash
Archived model identifier	deepseek:deepseek-v4-flash; model_id=deepseek_flash
Reasoning effort	xhigh
Context-compression threshold	150,000 tokens
Initial codebase	.gitignore and genesis.toml; no compiler implementation
Root-task inputs	Phase-I compiler blueprint; Phase-II continuation with handoff summary
Turn limits	2,048 root turns; 128 turns per non-root episode
Delegation and retry limits	Maximum depth 8; maximum retries 15
External test sources	c-testsuite, LLVM test suite, LZ4 and SQLite
Generated-program tests	Csmith through the committed harness
Rust toolchain	Rust stable; edition 2024; minimum Rust version 1.85; rustfmt and clippy
Compiler target	C11 primary and C23 stretch; x86 and x86-64 mandatory; AArch64 optional

Table S8. Machine and software environment for the compiler experiments.

Item	Recorded value
CPU	AMD Ryzen 7 PRO 6850HS with Radeon Graphics
CPU layout	x86-64; one socket; 8 physical cores; 2 threads per core; 16 online logical CPUs
Memory	64 GB
CPU frequency	403.7300–4787.0820 MHz; frequency boost enabled
Cache	256 KiB L1d, 256 KiB L1i, 4 MiB L2 and 16 MiB L3 in aggregate
NUMA	One node containing CPUs 0–15
Container and host	ArchLinux container; NixOS-built Linux 6.18.39 host kernel
Filesystem	ZFS with copy-on-write worktrees
Tool isolation	systemd-run and cgroups for every agent tool call
Missing reproduction fields	Container digest, package lock, per-tool cgroup limits, exact Rust/linker revisions and host-load traces

The compiler-formation and continuation runs used the same dedicated machine. This removes machine identity as a difference between the two continuation paths, but concurrency, CPU frequency, agent counts and token use still differed.

9.9 ARCHIVE COMPLETENESS

The archive status is **completed**. Four records with missing parent IDs remain in the raw count but are excluded from the parent-child tree. Some provider, message and environment details are missing, so an exact replay is not possible from this package alone. These missing fields do not change the reported final repository and test measurements.

10 CONTINUITY: DEVELOPMENT ACROSS FOUNDATION-MODEL REPLACEMENT

10.1 SHARED STARTING WORLD

This experiment uses a different compiler history from the DeepSeek formation run in Section 9. It has three stages: initial development with GLM 5.2, continuation with GLM 5.2 and continuation from the same completed compiler with DeepSeek V4 Flash. Both continuation runs start from commit 37216cfa254a. No result from the DeepSeek formation run is used as a continuation baseline.

Both continuation runs received the same completed GLM repository, saved project context, user instruction, test families and recorded controller limits. The instruction was: *“Continue and finish all remaining work, achieve 100% pass rate, excluding csmith as it is not installed.”* Agent count, concurrency, token use, wall time and code growth were not fixed in advance.

10.2 EVALUATION PROTOCOL

Both continuations used Rust workspace tests, LLVM SingleSource, c-testsuite, LZ4 and SQLite at -O0. The saved LLVM case lists differ between snapshots, so the reported fractions are not results on one fixed test set.

10.3 RUN SETTINGS, LINEAGE AND RESULTS

There was one completed run for each continuation path, so the results describe these runs rather than a repeatable model effect.

Both continuations used the same recorded limits for depth, retries, turns and context compression, and both ran on the machine in Table S8.

Table S9. Compiler-continuation starting points and run settings.

Setting	Initial development	GLM continuation	DeepSeek continuation
Model path	New project → GLM 5.2	GLM compiler → GLM 5.2	Same GLM compiler → DeepSeek V4 Flash
Starting state	Empty tracked project	Completed compiler at 37216cfa254a	Same completed compiler at 37216cfa254a
Task	Build new compiler	Continue existing compiler	Continue existing compiler
Root role	Not recorded	Manager	Manager
Depth and retry limits	8; 15	8; 15	8; 15
Turn limits	2,048 root; 128 delegated	2,048 root; 128 delegated	2,048 root; 128 delegated
Compression threshold	150,000 tokens	150,000 tokens	150,000 tokens
Test families	Rust, LLVM, c-testsuite, LZ4, SQLite	Same families	Same families
Csmith	Requested; unavailable	Excluded	Excluded
Runs	1	1	1
Shared fixed budget	Not specified	Not specified	Not specified

Table S10. Run summary for compiler continuation.

Metric	Initial GLM	GLM continuation	DeepSeek continuation
Elapsed time (h)	136.56	21.99	17.10
Spawned Agents	562	98	178
Archived records	504	97	168
Archive coverage	89.7%	99.0%	94.4%
First-parent commits	619	88	31
Maximum observed depth	5	4	8
Peak active Agents	21	9	19
Mean active Agents	2.86	2.94	5.70
Summed Agent-hours	390.57	64.65	97.57
Median duration (min)	14.4	16.1	17.0
90th-percentile duration (min)	83.2	59.8	70.2
Code-changing records	479	90	160
No-change records	25	7	8

DeepSeek finished earlier, but it also used more agents, more concurrent episodes, more archived agent-hours and more tokens. The wall-time difference is therefore not a speed comparison under equal resources.

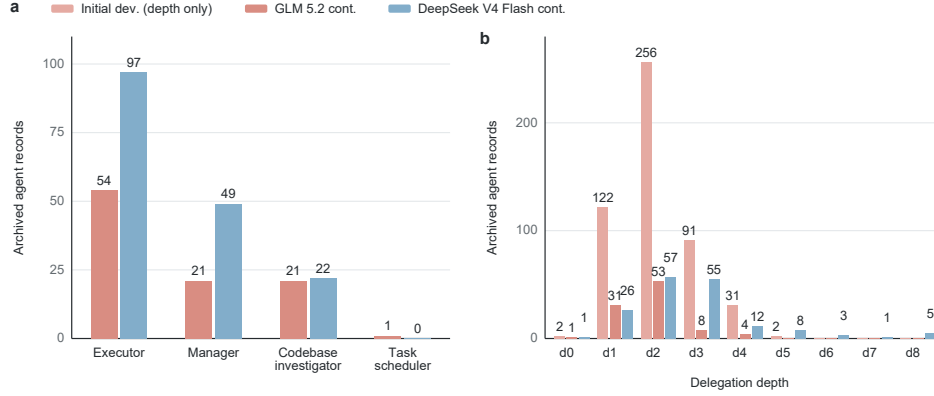
Table S11. Archive coverage for compiler continuation.

Stage	Spawned	Archived	Coverage	Missing-parent issue
Initial development	562	504	89.7%	Six absent parent IDs referenced by 15 records
GLM continuation	98	97	99.0%	Two records reference absent parent 562
DeepSeek continuation	178	168	94.4%	No missing-parent reference among archived records

Missing archived records can make reconstructed depth, duration, parent links and concurrency incomplete. The parent–child and organization statistics in this section therefore describe the archived records, not every spawned agent.



Supplementary Fig. S7. Delegation trees for compiler continuation. The upper tree shows the 504 archived records from initial GLM development, including two root segments and depth to d5; agent roles were not recorded for this stage. The lower trees show the 97 archived GLM 5.2 continuation records and 168 archived DeepSeek V4 Flash continuation records. Distance from the centre shows delegation depth, node shape shows the recorded role where available and colour shade shows completion order.



Supplementary Fig. S8. Agent roles and delegation depth in compiler continuation. **a**, Role counts for the 97 archived GLM 5.2 continuation records and 168 archived DeepSeek V4 Flash records; roles were not recorded for initial development. **b**, Delegation-depth counts for initial development and both continuations. These counts describe archived records, not all spawned agents.

10.3.1 RESOURCES, REPOSITORY GROWTH AND VALIDATION

Prompt-cache reuse was high in both continuation runs, but DeepSeek used substantially more total tokens than GLM. The two runs therefore cannot be compared as equal-budget efficiency tests.

Table S12. Token use and cost in compiler continuation.

Metric	Initial GLM	GLM continuation	DeepSeek continuation
Input tokens	2,245,871,926	543,563,883	902,775,744
Cached input tokens	2,163,779,584	534,150,656	884,375,552
Fresh input tokens	82,092,342	9,413,227	18,400,192
Cached input/input	96.34%	98.27%	97.96%
Output tokens	19,334,886	3,702,697	8,704,733
Total tokens	2,265,206,812	547,266,580	911,480,477
Input cost (US\$)	677.511916	152.057688	5.052314
Output cost (US\$)	85.073659	16.291867	2.437395
Total cost (US\$)	762.585575	168.349555	7.489709

The GLM-continuation archive stores zero in its cost fields; we treat those entries as missing. The US\$168.349555 GLM continuation cost is reconstructed from the rate schedule that reproduces the recorded initial-development GLM bill. The initial-development and DeepSeek costs are recorded in the archive. We therefore do not compare cost efficiency between the two models.

Table S13. Tests reported at the end of compiler continuation.

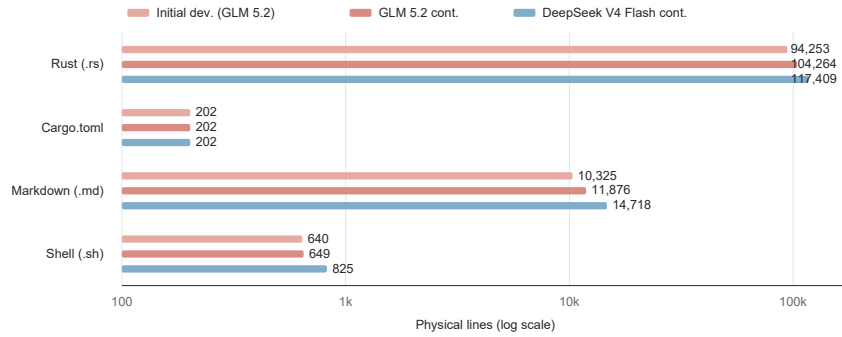
Test target	Initial development	GLM continuation	DeepSeek continuation
Rust unit tests	1,136	1,226 (+90)	1,350 (+214)
LLVM SingleSource	1,558/1,870 (83.3%)	1,445/1,448 (99.79%)	1,820/1,820 (100%)
c-testsuite	220/220	220/220	220/220
LZ4	4/4 files	4/4 files	4/4 files
SQLite at -O0	Basic run passed	Compiled	Compiled, linked and ran
Csmith	Unavailable	Excluded	Excluded

The LLVM case lists differ among snapshots, so each fraction applies only to its own saved list. No cause is assigned to the three non-passing GLM-continuation cases because per-case diagnostics are unavailable. The c-testsuite, LZ4 and SQLite values come from saved completion records and were not rerun while preparing this Supplementary Information.

Table S14. Physical line counts for the compiler-continuation snapshots.

Included file type	Task 1	GLM 5.2	DeepSeek V4 Flash
Rust	94,253	104,264	117,409
Cargo.toml	202	202	202
Markdown	10,325	11,876	14,718
Shell	640	649	825
Total	105,420	116,991	133,154

The initial-development total in Table S14 belongs to the separate GLM compiler history and counts only tracked Rust, Cargo.toml, Markdown and shell files. The 248,989-line total in Section 9 comes from a different repository and counts all tracked text files. The two numbers are therefore not directly comparable.



Supplementary Fig. S9. Physical line counts during compiler continuation. Tracked Rust, Cargo.toml, Markdown and shell lines are shown for the initial GLM compiler and the two continuation results. The horizontal axis is logarithmic so that small project-control files remain visible. Counts include blank lines.

Both GLM 5.2 and DeepSeek V4 Flash extended the same completed GLM compiler world. This shows continuation after agent turnover and after model replacement in the observed runs. It does not show why continuity was possible, whether the non-code records were necessary, or which model is better.

11 REDEVELOPMENT: MESA FROM FORTRAN TO RUST

11.1 STARTING SYSTEM AND RUN SETTINGS

The experiment used a lightly modified MESA fork (upstream: <https://github.com/MESAHub/mesa>) at commit 461dcba94f33a0dd991db129d406a877fcc9dfdc as a read-only reference. The Rust output is archived at commit 2ad071e658f8d4de21aa9f0e906592c364e10023. The run used DeepSeek V4 Flash with xhigh reasoning effort and a 150,000-token context-compression threshold.

This was one run. A second root-level objective began at 31.720 h after a handoff from the first. Resource totals use the top-level archive record because parent records include usage from their descendants.

The root objective requested a broad Rust rewrite with API compatibility, testing and performance optimization. The results reported here cover only the 13 mapped foundation and physics modules and the workloads below. They do not establish a complete Rust replacement for MESA.

Table S15. MESA-to-Rust run and timing settings.

Quantity	Value
Model	DeepSeek V4 Flash
Model settings	xhigh reasoning; 150,000-token compression threshold
Recorded agent roles	Codebase lead, manager, executor and codebase investigator
Controller limits	Maximum depth 8; retries 15; 2,048 root turns and 128 child turns
Source baseline	Modified MESA fork, commit 461dcba94f33
Timing host	Shared Intel Xeon Platinum 8336C system; 64 physical cores / 128 hardware threads; benchmark processes pinned to CPUs 0–3
Timing runs	25 direct-binary runs per workload after warm-up; separate 40-run burn-proxy check
Fortran build	gfortran 12.2.0; -O3 -march=native -ffp-contract=fast -std=f2008
Rust build	opt-level=3; fat LTO; codegen-units=1; panic=abort

Table S16. MESA-to-Rust resource use and archive coverage.

Quantity	Recorded value
Experiment interval	3–5 August 2026 (UTC)
Elapsed wall time	33.219 h
Root-agent handoff	31.720 h
Spawned agents	272
Archived agent records	260
Archive coverage	95.6%
Input tokens	771,755,551
Cached input tokens	744,102,016
Cached input share	96.4168%
Output tokens	16,721,326
Total tokens	788,476,877
Recorded model-token cost	US\$10.636892

Archive coverage is the number of archived records divided by the top-level spawned-agent count. Any record-level statistics below therefore describe the available archive, not all spawned agents.

11.2 MIGRATION SCOPE AND MODULE MAPPING

The comparison maps 13 MESA module directories one-to-one to 13 mesa-rs crates. These MESA directories contain **139,414 physical Fortran lines**, including each module’s local test/ programs. The matching Rust crates contain **67,373 library lines** and **19,955 crate-local test lines**, or **87,328 mapped Rust lines** in total. A separate count of all .rs files in the workspace gives **89,946 lines**.

Summing the archived root-level components gives 89,945, one line fewer; we keep this one-line discrepancy visible. It does not affect the mapped total of 87,328 lines. All counts include comments and blank lines and measure source size, not feature equivalence.

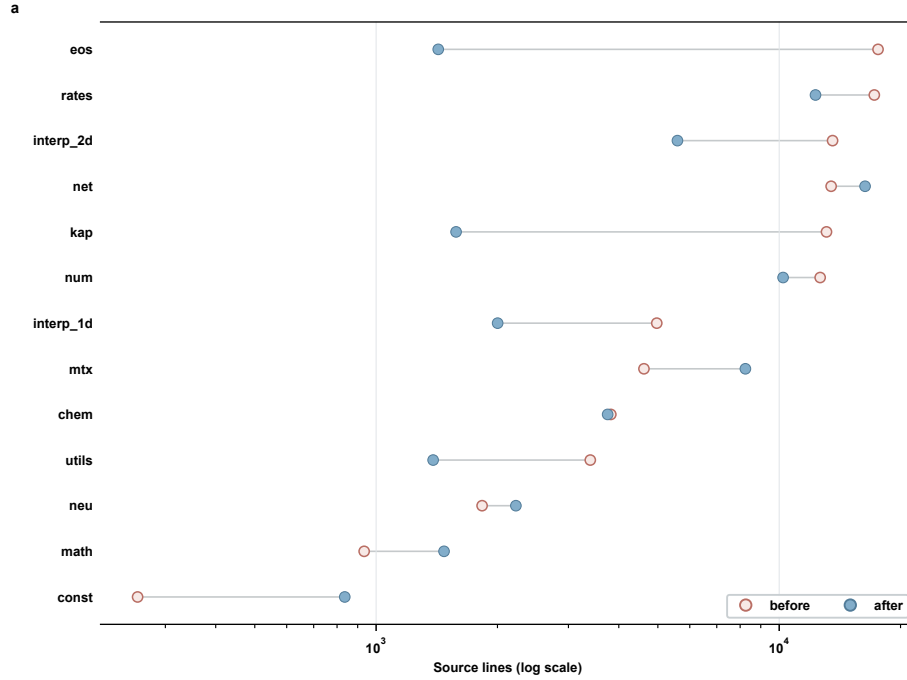
These ratios compare physical line counts only: 139,414 Fortran lines versus 67,373 Rust library lines gives $2.07\times$, and versus 87,328 mapped Rust lines including crate-local tests gives $1.60\times$. The modules vary widely. In particular, the Rust eos and kap crates cover narrower functionality than the full MESA modules, so their smaller line counts should not be read as a Rust-versus-Fortran compression result.

Table S17. Module-level source line counts for MESA and mesa-rs.

Layer	Module	Fortran LOC	Rust lib	Rust tests	Rust total	Fortran/Rust
Foundation	const	308	836	90	926	0.33
Foundation	utils	3,569	1,384	370	1,754	2.03
Foundation	math	1,020	1,474	807	2,281	0.45
Foundation	mtx	5,206	8,244	1,728	9,972	0.52
Foundation	interp_1d	5,851	2,002	937	2,939	1.99
Foundation	interp_2d	16,176	5,592	1,300	6,892	2.35
Foundation	num	18,357	10,225	3,078	13,303	1.38
Physics	chem	4,424	3,753	900	4,653	0.95
Physics	rates	18,547	12,304	4,975	17,279	1.07
Physics	neu	2,263	2,222	995	3,217	0.70
Physics	net	17,799	16,332	2,429	18,761	0.95
Physics	eos	25,817	1,426	1,514	2,940	8.78
Physics	kap	20,077	1,579	832	2,411	8.33
All	Total	139,414	67,373	19,955	87,328	1.60

The final column compares physical line counts; it is not a language-efficiency score. The 1,030 static #[test] annotations, 19,955 crate-local test source lines and 1,052 passing workspace tests are different measures and should not be interchanged.

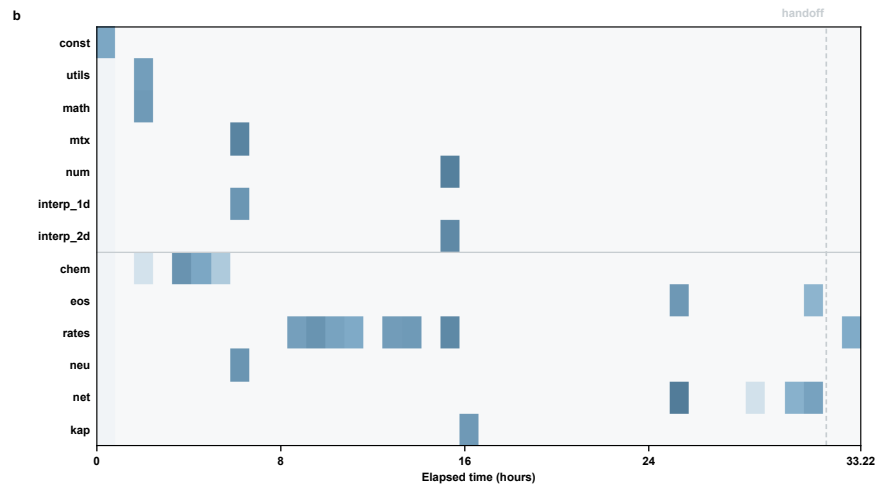
The migration does not cover several higher-level or broader parts of MESA, including the star, astero, binary, adipls, stella and gyre engines, the full seven-source EOS blend, the complete REACLIB dataset, single-precision _sg variants and auto_diff. The result is a migration of a core numerical and physics module chain, not a complete Rust replacement of MESA.



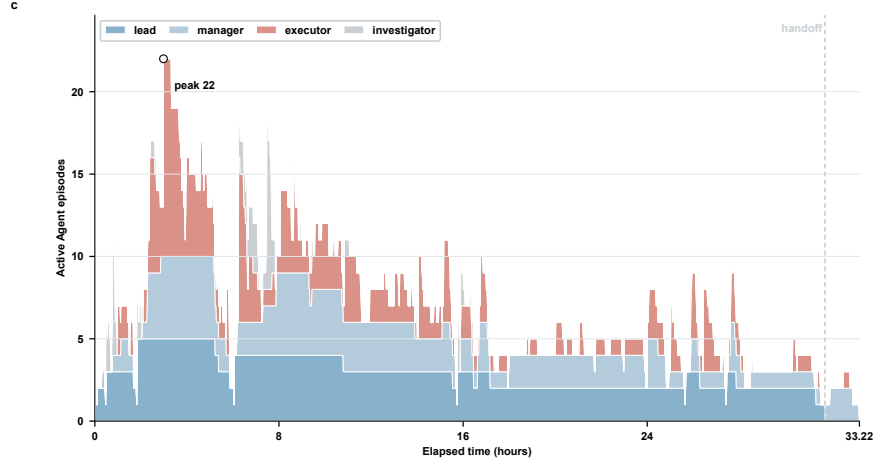
Supplementary Fig. S10. Module-level source line counts for MESA and Rust. For each of the 13 mapped modules, physical Fortran lines in the module directory, including local test programs, are compared with Rust library lines and Rust library-plus-crate-test lines. The logarithmic axis shows both larger and smaller ports. Counts include comments and blank lines and do not imply feature equivalence.

11.3 RUN TIMELINE AND CONCURRENCY

The archived run lasted 33.22 h, with the root-level handoff at 31.72 h. Reconstructing episode times at one-minute resolution gives a maximum of 22 overlapping archived episodes. This is a count of active assignments, not CPU use.



Supplementary Fig. S11. Accepted Rust source changes over time. First-parent Git activity is shown for the 13 migrated crates during the 33.22-h run. Colour intensity represents $\log(1 + \text{added} + \text{deleted})$ lines. The dashed line marks the root-level handoff at 31.72 h. Blank intervals mean that no first-parent source change was accepted at the plotted resolution; analysis or testing may still have been active.



Supplementary Fig. S12. Concurrent agent episodes during migration. Active archived episodes are reconstructed at one-minute resolution and stacked by recorded role. The maximum overlap was 22 episodes. The plot uses archived start and finish times and does not show CPU use.

11.4 NUMERICAL VALIDATION AND RUNTIME

The six-workload comparison uses the median of 25 runs per implementation after warm-up. These workloads are selected numerical kernels or pipelines, not a complete MESA run. In particular, the “end-to-end burn” case is a single-zone, constant-density burn proxy. Its timing covers burn integration but not table loading or composition setup; it is not a full MESA star evolution run.

The Fortran and Rust timers were not identical. For net and kap, Fortran uses `system_clock` and Rust uses `Instant`; for interpolation, Newton, ROS2 and EOS, Fortran uses `cpu_time` and Rust uses `Instant`. The workloads are reported as single-threaded and CPU-bound, and the two languages use different build toolchains. The runtime ratios therefore describe these particular binaries on this host and benchmark setup; they are not a general Rust-versus-Fortran speed comparison.

Across the 25-run batch, the Rust median was lower for all six workloads, with Fortran/Rust ratios from $1.55\times$ to $6.87\times$. EOS lookup and Newton solve were bit-exact under the recorded checks; the other relative checksum differences ranged from 5.1×10^{-15} to 3.1×10^{-9} .

Table S18. Numerical agreement and runtime performance for six migrated workloads.

Workload	Fortran (s)	Rust (s)	Speedup	Checksum difference
End-to-end burn	0.446	0.287	$1.55\times$	3.1×10^{-9}
EOS lookup	1.786	1.115	$1.60\times$	Bit-exact
Opacity lookup	0.532	0.269	$1.98\times$	1.3×10^{-13}
2D interpolation	0.159	0.101	$1.58\times$	4.9×10^{-12}
ROS2 integration	0.722	0.136	$5.30\times$	5.1×10^{-15}
Newton solve	0.084	0.012	$6.87\times$	Bit-exact

A separate timing check repeated the burn proxy 40 times per implementation after three warm-up runs, again pinned to CPUs 0–3. The median was 0.2996 s for Fortran and 0.2427 s for Rust, a $1.23\times$ ratio. Thirty of the 40 Rust timings were below the Fortran median; this is not a 30/40 paired win rate because the paired run vectors are unavailable. Both implementations reported the same integrator counts: 8,889 function evaluations, 199 Jacobian evaluations, 199 steps, 198 accepted steps and one rejected step. The abundance-checksum relative difference was 3.1×10^{-9} .

Table S19. Separate 40-run timing check for the burn proxy.

Statistic	Fortran (s)	Rust (s)	Fortran/Rust
Minimum	0.2897	0.2372	1.22×
P10	0.2922	0.2385	1.22×
P25	0.2927	0.2392	1.22×
Median	0.2996	0.2427	1.23×
Mean	0.3447	0.2735	1.26×
P75	0.4416	0.3609	1.22×
P90	0.4458	0.3729	1.20×
Maximum	0.4490	0.3815	1.18×

The 1.55× burn ratio in the six-workload table and the 1.23× ratio in the separate 40-run check come from different batches and should not be pooled. The package contains summary statistics for the 40-run check but not the 80 individual timings, so we cannot reconstruct a confidence interval or paired test. Timing was also sensitive to host load: one noisy interleaved run gave a ratio of 0.82×, while a quieter rerun gave about 1.22×. We therefore use the 1.23× 40-run median as the more conservative burn result and do not claim a general speed advantage.

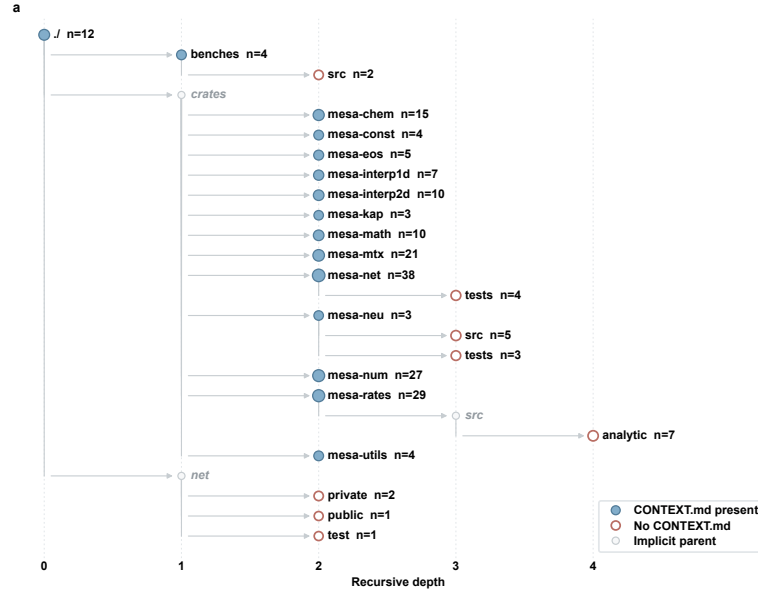
11.5 DEPENDENCY STRUCTURE AND RECURSIVE CONTEXT

The mapped dependency graphs contain 142 MESA provider-to-dependent edges and 41 `mesa-rs` crate edges. Forty edges appear in both graphs, 102 appear only in the MESA graph and one appears only in Rust. A MESA-only edge means only that no matching direct Rust crate edge was found; it does not by itself show missing or removed functionality.

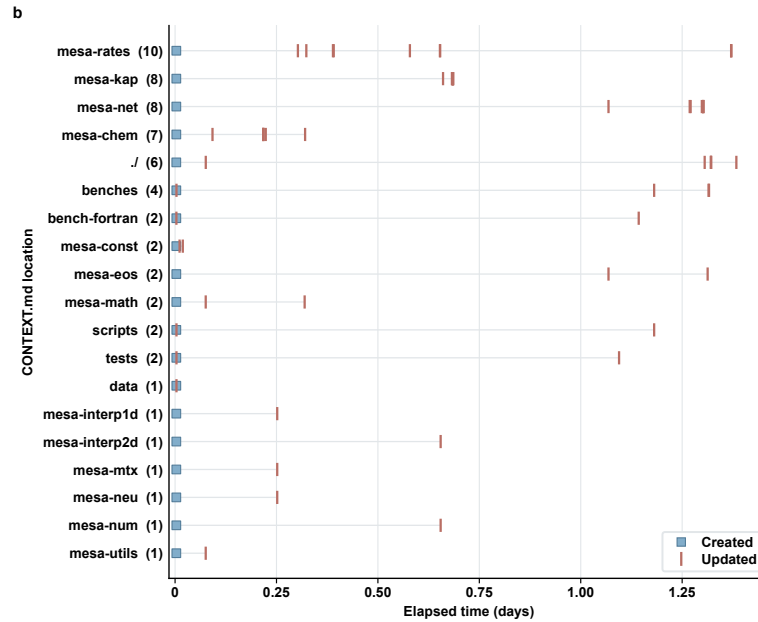
Table S20. Mapped dependency counts for MESA and `mesa-rs`.

Category	Edges	Interpretation
MESA	142	Provider-to-dependent edges parsed from <code>INTERNAL_DEPENDS_ON</code>
<code>mesa-rs</code>	41	Mapped crate dependencies parsed from <code>Cargo.toml</code>
Retained	40	Edges present in both mapped graphs
MESA-only	102	No matching direct Rust crate dependency in the mapped graph
Rust-only	1	Dependency introduced in <code>mesa-rs</code>

Archived assignments contain 34 observed context paths; adding parent paths needed to connect the hierarchy gives 37 nodes. Twenty-six observed paths contain `CONTEXT.md`. Git history records 26 file creations and 62 later accepted updates affecting 19 files. This shows that shared context changed during the run, but the archive does not show which agent read which file or whether the updates improved performance.



Supplementary Fig. S13. Recursive context paths. The hierarchy shows context paths found in archived assignments, with parent paths added where needed to connect the tree. Markers show whether a `CONTEXT.md` file was present. These paths record task context, not directory traversal or file access.



Supplementary Fig. S14. Changes to shared context files. First-parent Git creation and update events are shown for `CONTEXT.md` files during migration. The timeline shows when shared context changed; it does not show which agents read each file or whether an update improved performance.

The run produced 13 Rust crates for the reported MESA core-module scope. The final workspace passed 1,052 tests, and the tested numerical workloads showed the agreement reported above.

12 EVIDENCE BOUNDARIES AND AUDIT NOTES

12.1 WHAT EACH EXPERIMENT ESTABLISHES

Table S21 states what each experiment supports and what remains unresolved. Detailed measurements are in Sections 9–11.

Table S21. What each experiment supports and what remains unresolved.

Experiment	What else could explain the result	Evidence in this study	What the result supports
Compiler formation	The task specification and model knowledge may explain part of the resulting organization	Repository with no compiler implementation, a 1,015-record reconstructed parent–child tree, repository counts and seven test families	One observed run formed and broadly tested a working compiler; no repeated mechanism test
Compiler continuation	Model differences, unequal resource use or the source code alone may explain the observed continuation	Same saved GLM starting world and instruction, but one run per path, different resource use, different LLVM case lists and no code-only or fresh-agent control	The same completed compiler was continued with GLM 5.2 and DeepSeek V4 Flash
MESA migration	Limited module and workload coverage and the timing setup may explain part of the result	Thirteen mapped module directories, 13 Rust crates, six 25-run workload summaries and a separate 40-run burn-proxy summary	Numerical agreement on the tested workloads and lower measured runtimes under the reported setup

12.2 OBSERVED FAILURES AND MISSING RECORDS

Table S22 lists observed failures and missing records. We do not assign a cause when the available evidence does not show one.

Table S22. Observed failures, missing records and how they are handled.

Experiment	Observed issue	How it is handled
Compiler formation	Four direct missing-parent records; four of 36 reported LLVM cases did not pass; seven of 100 Csmith seeds were skipped	Rows with missing parents are excluded only from the parent–child tree; non-passing and skipped test cases remain in the reported denominators
Compiler continuation	GLM 5.2 passed 1,445/1,448 on its retained LLVM manifest	The exact count is reported. No cause is claimed for the three non-passing cases because per-case diagnostics are unavailable
Continuation archives	Archive coverage is 89.7% for initial GLM development, 99.0% for GLM continuation and 94.4% for DeepSeek continuation	Parent–child, role, depth and concurrency summaries use only the archived records available for each stage
MESA numerical checks	Four of six tested workloads have small non-zero checksum differences; the module and workload coverage is incomplete	The numerical differences and coverage limits are reported directly; no claim of complete MESA equivalence is made
MESA timing checks	One noisy interleaved end-to-end timing run favoured Fortran (0.82× Rust/Fortran speed ratio), whereas the dedicated 40-run summary favoured Rust at 1.23× median	Both observations are reported; runtime claims are limited to the reported host and timing setup and are not generalized to Rust versus Fortran

Experiment	Observed issue	How it is handled
MESA line counts	Independent workspace counting paths differ by one physical Rust line (89,946 versus 89,945)	The discrepancy is reported and is not used in the main comparison; the mapped 13-crate total of 87,328 lines is the comparison value

12.3 CAUSAL TESTS NOT PERFORMED IN THIS STUDY

The present experiments establish observed capabilities under the reported settings, but they do not isolate the causal contribution of every persistent record or recursive mechanism. One direct test would hold executable code fixed while changing accepted non-code development records.

Here, $\text{Dev}(v)$ means the accepted non-code records stored with version v that can affect later work. It is a way to separate those records from executable code for an experiment; it is not a new object in the model.

To show that these non-code records matter beyond the executable code, one would need two accepted versions v^A and v^B with identical executable code but different accepted non-code records, and then give both the same future task u :

$$\begin{aligned} \text{Code}(v^A) &= \text{Code}(v^B), & \text{Dev}(v^A) &\neq \text{Dev}(v^B), \\ P(Y \mid (v^A, p), u, B) &\neq P(Y \mid (v^B, p), u, B), \end{aligned} \tag{9}$$

with the same path, model, tools, evaluator and resource budget B . We did not run this experiment in the present study.