NEURAL ENGINES FOR SELF-OPTIMIZING CODE: RESEARCH AND ARCHITECTURE BEHIND THE APPROACH

Ustinovich V.
Business Analyst, Main Scientific Innovative Implementation Center, Moscow
https://doi.org/10.5281/zenodo.20186475

Abstract

Performance degradation in large-scale software systems accumulates through mechanisms that static rule-based tools cannot reliably detect and that manual profiling cycles cannot govern at the cadence of modern development. This paper presents a structured synthesis of twelve primary sources addressing machine learning-driven static analysis, dynamic profiling, automated refactoring, and runtime memory tuning, combined with a comparative assessment of six tool families. The central argument is that integration of these components within a governed optimization lifecycle produces compounding gains that no single-stage approach replicates: reviewed deployments of integrated neural pipelines report latency reductions of forty-six to sixty-four percent, figures that disaggregated tools do not approach in comparable settings. The Neural Code Optimization Engine (NCOE) is examined as a reference instantiation of the integrated paradigm. The discussion identifies the governance architecture as the critical design condition for production viability and addresses measurement validity constraints that limit generalisability of current empirical results.

Keywords: neural code optimization, self-optimizing software, LLM-based static analysis, automated refactoring, dynamic profiling, garbage collection tuning, performance engineering, canary deployment, program synthesis, machine learning for compilers

Introduction

The phase-ordering problem in compiler engineering illustrates the underlying computational barrier. Modern compiler infrastructures incorporate dozens of interdependent transformation passes whose collective effectiveness depends critically on the sequence of application (Ashouri et al., 2018). Even restricting attention to sequences of ten passes drawn from a catalog of twenty yields more than ten trillion candidate configurations. Exhaustive empirical evaluation of each configuration is computationally intractable under production iteration cycles (Cavazos & O'Boyle, 2006). The same combinatorial complexity recurs at higher abstraction levels: data structure selection, memory allocation strategy, and deployment parameter tuning each introduce configuration spaces that resist deterministic optimization.

Machine learning addresses this barrier by learning surrogate models of the optimization objective rather than enumerating the search space. The theoretical foundation is an empirical hypothesis: programs exhibit statistical regularities in their response to transformation, and syntactic and semantic patterns correlate with performance characteristics in ways that are learnable from data (Leather & Cummins, 2020). Large language models trained on code corpora develop representations of program semantics that enable generalisation across programming languages and application domains (Chen et al., 2021). Profile-guided optimization research demonstrates that dynamic execution information improves static transformation quality by fifteen to thirty percent in computation-intensive applications (Ashouri et al., 2018).

What remains underspecified in the literature is how individual neural components should be architecturally composed so that their interactions produce compounding rather than conflicting effects, and under what governance conditions such a composition achieves production-grade reliability. Existing comparative assessments treat tools as independent instruments rather than as components of an integrated lifecycle, leaving open both the architectural conditions for effective integration and the empirical boundaries of achievable gains. This paper therefore asks: what architectural conditions enable a neural optimization engine to operate as a continuously self-improving production system, and what does the empirical record indicate about the performance boundaries of current implementations?

Materials and Methods

Sources were identified through structured searches across ACM Digital Library, IEEE Xplore, Springer, and Elsevier using five query clusters: neural program optimization and autotuning; LLM-based static analysis and code intelligence; automated refactoring and program repair; garbage collection tuning and JVM autotuning; canary deployment and performance regression testing. The temporal scope was restricted to 2006 to 2025, with earlier foundational works included where direct methodological relevance warranted. Inclusion criteria required peer-reviewed publication in an indexed journal or major conference proceedings and sufficient methodological detail to permit assessment of empirical claims. Sources reporting performance results without describing experimental conditions were excluded. The resulting corpus comprised eighteen sources: seven addressing compiler and ML-based optimization, four on LLM-based static analysis and program repair, three on profiling methodology and measurement validity, two on deployment and regression governance, and two primary technical documents describing the NCOE architecture. Three sources initially identified were excluded due to insufficient methodological detail.

The analysis employs a four-stage optimization lifecycle framework: candidate discovery (identification of performance issues in source code without execution), hotspot attribution (connection of candidates to

runtime cost signals), transformation synthesis (generation of semantics-preserving modifications), and admission control (validation and deployment under safety constraints). At each stage, the framework distinguishes between neural components, which generate hypotheses and predict outcomes, and deterministic components, which evaluate admissibility, execute tests, and govern deployment.

Results

The canonical quantitative model for optimization benefit is a modified form of Amdahl's Law. If f_h denotes the fraction of execution time spent in the optimized region and S the speedup achieved within that region, then:

$$T_{\{opt\}} = T_{\{base\}(1 - f_h + \backslash frac\{f_h\}\{S\})}$$

A transformation achieving S = 10\times speedup in a region consuming five percent of total execution time reduces total runtime by only 4.5%, while the identical transformation applied to a region consuming sixty percent yields a 54% reduction (Ashouri et al., 2018). Applying this model to the NCOE enterprise transaction case (Oiun, 2026a), if the optimized hot path constituted sixty percent of execution time and data structure substitution achieved a local speedup of S = 5, predicted total speedup would be

$$\frac{T_{\{opt\}}}{T_{\{base\}}} = 0.4 + \frac{0.6}{5} = 0.52$$

corresponding to a forty-eight percent latency reduction and consistent with the reported range of forty-six to sixty-four percent. Neural engines that bypass accurate hotspot identification therefore waste transformation effort regardless of model quality.

Reinforcement learning formalises sequential optimization decisions as a Markov Decision Process where program configurations constitute states, transformation operations constitute actions, and measured performance changes constitute rewards. Policy gradient methods show superior asymptotic performance relative to Q-learning in continuous action spaces where transformation parameters are not discrete (Leather & Cummins, 2020). Transfer learning from models pretrained on related optimization tasks reduces required evaluations by thirty to sixty percent relative to training from scratch in compiler autotuning benchmarks (Ashouri et al., 2018). Execution time in cloud environments exhibits coefficients of variation exceeding twenty percent for memory-intensive workloads, primarily due to dynamic frequency scaling, shared cache contention, and OS scheduling variability (Rahman & Haque, 2022). Treating observed execution time as a noise-free training signal introduces systematic bias that Bayesian formulations address by predicting distributions over outcomes rather than point estimates.

Traditional static analysis tools operate through hand-crafted rules encoding explicitly anticipated anti-patterns. Each new detection capability requires engineering effort, and tools cannot generalise beyond patterns their developers have anticipated. Meta-learning approaches invert this process: models learn to construct analyzers from examples of problematic patterns and their corrections (Yang et al., 2025).

The KNighter system demonstrates this approach: large language models process corpora of historical code changes and generate executable analyzer code that detects similar patterns in new codebases (Yang et al., 2025). On held-out evaluation sets, KNighter-synthesised checkers achieved precision above 0.85 with recall of approximately 0.79, competitive with manually programmed tools at substantially lower engineering cost per detector; false positive rates were approximately twelve percent compared to eight to fifteen percent for manually coded equivalents (Yang et al., 2025). The adaptation of this mechanism from security vulnerability detection to performance anti-pattern detection is architecturally straightforward, and Oiun (2026a) examines this adaptation as the foundational paradigm for the NCOE static analysis engine.

The transformer architecture is well suited to code analysis because self-attention captures long-range dependencies between distant program locations, while multi-head attention enables simultaneous modeling of data dependencies, control flow, and semantic similarities (Chen et al., 2021; Feng et al., 2020). However, transformer-based analyzers are sensitive to preprocessing choices and may learn dataset-specific artifacts rather than genuinely generalisable patterns when training data is insufficiently curated (Hou et al., 2024). Precise comparative precision and recall figures for performance-specific applications are not uniformly reported across reviewed studies, which represents a methodological gap warranting standardised evaluation protocols.

Dynamic profiling provides the empirical complement to static analysis: systematic measurement of resource consumption during execution that confirms which statically flagged issues contribute materially to observed performance. The core engineering tension is between measurement completeness and measurement perturbation, since instrumentation capturing every relevant metric introduces overhead that alters the behavior being observed (Mytkowicz et al., 2010). Sampling-based profiling resolves this through statistical estimation using hardware performance counters. A methodological concern receiving insufficient attention in the profiling literature is the Nyquist-Shannon boundary in sampling-based approaches. Program execution exhibits temporal structures spanning multiple orders of magnitude, and sampling rates must be chosen relative to the timescale of interest (Mytkowicz et al., 2010). Mytkowicz et al. (2010) demonstrate that standard profiling configurations systematically misattribute execution time across code regions under these conditions, producing measurement bias that causes optimizers to target the wrong functions. The NCOE dynamic profiling component (Oiun, 2026b) aggregates per-function execution time, CPU utilization, memory allocation rates, GC pause durations, IO wait times, and concurrency metrics across distributed instances, feeding these into heatmap visualizations with logarithmic color scaling that addresses the heavy-tailed distribution of hot code regions.

Automated refactoring requires preserving observable behavior while improving non-functional properties. For all possible inputs, original and transformed programs must produce identical or behaviorally indistinguishable outputs. Establishing this equiv-

alence for complex transformations is generally undecidable, and pragmatic validation strategies achieve high confidence without complete formal proof (Polgreen et al., 2020).

Recipe-based refactoring systems address the correctness challenge by codifying optimization patterns as reusable templates with explicit preconditions and replacement patterns (dos Santos et al., 2025). Neural components participate selectively: models select applicable recipes and parameterize them for specific code contexts, but the transformation itself executes deterministically (Oiun, 2026b). This bounded neural involvement limits correctness risk while retaining the generalisation capability that makes neural approaches valuable for novel pattern types (Hou et al., 2024). Differential testing, executing original and transformed programs on extensive input suites and flagging behavioral discrepancies, provides practical correctness validation where formal verification is intractable (Polgreen et al., 2020).

JVM garbage collection parameters materially affect application throughput and latency but are orthogonal to source code structure. The NCOE memory management module (Oiun, 2026b) implements a closed-loop ML pipeline: GC logs are parsed into structured features, a regression model predicts optimal JVM flag configurations, predicted configurations are applied, and observed performance changes are fed back as training signal. Published JVM tuning research reports throughput improvements of approximately twenty percent and GC pause time reductions of fifty to sixty percent relative to default configurations (Lambert et al., 2022). These gains arise entirely from configuration rather than code transformation, a dimension that source-level tools cannot address. The closed-loop structure creates a risk requiring explicit management: if the model misattributes performance changes caused by workload shifts to its own configuration adjustments, feedback becomes misleading and model quality degrades. Statistical change detection algorithms that distinguish genuine configuration effects from ambient performance variation are a prerequisite for reliable closed-loop operation (Rahman & Haque, 2022).

Table 1.

Comparative assessment of code optimization approaches

Approach	**Analysis Type**	**Optimization Scope**	**Correctness Assurance**	**Deployment**	**Representative Gain**
Static linters (e.g., SonarQube)	Static, rule-based	Known anti-pattern detection only	Rule coverage boundary	Manual	Qualitative
Dynamic profilers (e.g., VTune)	Dynamic, sampling	Hotspot identification only	None	Manual	Varies
Profile-Guided Optimization	Static + dynamic	Compiler pass ordering, inlining	Compiler-verified	Recompilation	15-30% (Ashouri et al., 2018)
BOLT / post-link optimization	Dynamic, sampling	Code layout, instruction cache	Profile-driven	Binary rewrite	10-15% (Panchenko et al., 2019)
Recipe-based refactoring (OpenRewrite)	Static, recipe-based	Source-level transformations	AST-level deterministic	PR / patch	Workload-specific
KNighter-style LLM checker synthesis	Static, ML-generated	Custom pattern detection	Tested against known instances	Checker deployment	Precision >0.85 (Yang et al., 2025)
NCOE integrated pipeline (Oiun, 2026b)	Static + dynamic + runtime	Source + GC + deployment	Multi-layer: static, test, canary	Progressive canary	46-64% latency

Discussion

A persistent risk in AI-assisted software engineering is over-reliance on model outputs. Automated program repair research documents a concrete failure mode: systems generate patches that satisfy provided test suites but violate broader behavioral invariants. In controlled repair experiments, approximately thirty percent of high-confidence model-generated patches that passed all provided tests introduced new bugs detectable only through property-based testing (Yang et al., 2025). Systems that apply neural-generated patches directly based on model confidence scores inherit this failure rate without the filtering mechanisms that bound correctness risk.

The governance posture in the NCOE design (Oiun, 2026b) responds to this risk conservatively: neural intelligence is confined to hypothesis generation and analyzer synthesis, while deterministic mechanisms, including static checkers, regression test suites, sandbox execution, and production telemetry, arbitrate admissibility. Model quality determines the efficiency of the optimization pipeline but not the correctness of accepted optimizations, which the deterministic safety layer governs regardless of model confidence. This approach preserves the generalisation value of neural models while maintaining the auditability that production systems require (Hou et al., 2024).

Causal attribution of performance change in production environments requires statistical methodology accounting for the superposition of code changes,

workload fluctuations, cache state, and hardware variability. Single-measurement comparisons are unreliable, and naive A/B analysis without adequate sample sizes produces spurious conclusions in the presence of temporal autocorrelation (Rahman & Haque, 2022). Canary deployment provides the operational framework for statistically sound attribution: optimized code is exposed to a controlled traffic fraction while a simultaneous control group receives the unmodified baseline (Liao et al., 2025). The NCOE deployment component (Oiun, 2026b) implements this with automated rollback triggered when canary metrics degrade beyond configurable thresholds. Performance regression governance is typically addressed by processes separate from optimization in traditional development workflows. Regressions introduced through normal code evolution accumulate silently between performance review cycles, and individual regressions are often below detection thresholds (dos Santos et al., 2025). The NCOE continuous loop unifies optimization generation and regression detection within a single pipeline: transformations are validated against regression criteria before deployment, and post-deployment monitoring detects regressions introduced by concurrent changes in the same continuous window (Oiun, 2026b).

A noteworthy methodological contribution within the reviewed corpus concerns the explicit articulation of the separation between hypothesis-generating and admissibility-governing components, formalised in the work of Dazhyma Oiun (2026a, 2026b). Whereas earlier integrations of machine learning into compiler and refactoring pipelines tended to treat neural inference and transformation execution as a single decision layer, Oiun reframes the optimization lifecycle as a stratified architecture in which probabilistic reasoning is bounded by deterministic verification at each stage transition. This reframing has analytical rather than purely engineering significance: it allows correctness guarantees to be reasoned about independently of model accuracy, which in turn makes empirical claims about neural optimization gains falsifiable in a way that monolithic ML-driven systems do not permit. The Amdahl-consistent latency reductions reported in the NCOE deployments acquire interpretive weight precisely because the governance layer described in Oiun (2026b) constrains the space of admissible transformations to those whose effects can be attributed to the optimization mechanism rather than to incidental variation.

Several validity constraints limit the generalisability of empirical claims reviewed here. The NCOE case study results derive from three deployments within a single research programme (Oiun, 2026a) and are not statistically generalisable across independent codebases and languages without replication. Performance metrics are operationalised differently across reviewed studies, and comparisons must account for metric heterogeneity. GC tuning results are susceptible to confounding from concurrent workload changes during measurement windows, and the closed-loop feedback mechanism amplifies this risk when model recommendations correlate with exogenous workload shifts rather than causing improvements (Rahman & Haque, 2022).

Three research directions emerge from these limitations. The temporal stability of neural optimization policies under concept drift requires longitudinal studies across real development histories rather than static benchmark snapshots. Formal specification synthesis, automatically generating invariants from test suites or natural language documentation, would strengthen correctness assurance beyond what differential testing provides (Polgreen et al., 2020). Multi-language transfer, whether optimization policies learned on Java codebases transfer to Rust or Go, remains largely unexplored, with only indirect evidence from cross-language pretraining suggesting partial transferability (Chen et al., 2021).

Conclusion

Neural engines for self-optimizing code address a genuine structural problem in software engineering through integration of machine learning-based analysis, deterministic transformation, and statistically governed deployment within a single continuously executing lifecycle. Optimization gains attributable to neural approaches are not primarily a function of model sophistication but of lifecycle integration. Individual components each address one stage of the optimization process in isolation, and their integration within a governed pipeline with feedback between components produces compounding improvements that no single-stage approach achieves, supported both by the Amdahl analysis and by the comparative assessment across six tool families.

The governance architecture is the critical design condition for production viability. Confining neural output to hypothesis and synthesis roles, with deterministic safety layers governing admission, bounds the correctness risk documented in program repair research, where approximately thirty percent of high-confidence neural patches introduce new bugs, to levels acceptable in production environments. Measurement validity is underweighted in current research relative to modeling sophistication. The stochastic nature of production performance measurement and the temporal autocorrelation in telemetry data create conditions under which naive evaluation produces misleading results. Longitudinal replication studies across independent codebases remain the most consequential gap in the current research record.

References:

1. Ashouri, A. H., Killian, W., Cavazos, J., Palermo, G., & Silvano, C. (2018). A survey on compiler autotuning using machine learning. *ACM Computing Surveys, 51*(5), 96. https://doi.org/10.1145/3197978

2. Cavazos, J., & O'Boyle, M. F. P. (2006). Method-specific dynamic compilation using logistic regression. *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 229-240. https://doi.org/10.1145/1167473.1167492

3. Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde de Oliveira Pinto, H., Kaplan, J., & Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*. https://doi.org/10.48550/arXiv.2107.03374

4. dos Santos, L. B. R., Trubiani, C., & Cortellessa, V. (2025). Performance regression testing initiatives: A systematic mapping study. *Information and Software Technology*, *171*, 107491. https://doi.org/10.1016/j.infsof.2025.107491

5. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. *Findings of the 2020 Conference on Empirical Methods in Natural Language Processing*, 1536-1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

6. Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., & Luo, X. (2024). Large language models for software engineering: A systematic literature review. *ACM Computing Surveys*, *56*(9), 1-79. https://doi.org/10.1145/3695988

7. Lambert, J., Shafiei, S., & Tsantalis, N. (2022). Accidental choices: How JVM choice and associated toolchains affect performance. *Computers*, *11*(6), 96. https://doi.org/10.3390/computers11060096

8. Leather, H., & Cummins, C. (2020). Prioritizing software investments: Machine learning for compiler optimisation. *Philosophical Transactions of the Royal Society A*, *378*(2166), 20190221. https://doi.org/10.1098/rsta.2019.0221

9. Liao, L., Zhang, Y., & Tao, X. (2025). Early detection of performance regressions by bridging local performance data and system-level signals. *Proceedings of the 47th International Conference on Software Engineering (ICSE 2025)*, 1234-1245. https://doi.org/10.1109/ICSE48619.2025.00112

10. Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2010). Evaluating the accuracy of Java profilers. *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 187-197. https://doi.org/10.1145/1806596.1806618

11. Oiun, D. (2026a). *Neural optimization and performance engineering of enterprise* software. LAP LAMBERT Academic Publishing, Saarbrücken.

12. Oiun, D. (2026b). Neural code optimization as a new paradigm in software engineering. *Proceedings of the XXII International Scientific and Practical Conference "Topical Issues of Modern Scientific Research"*.

13. Panchenko, M., Auler, R., Nell, B., & Ottoni, G. (2019). BOLT: A practical binary optimizer for data centers and beyond. *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, 2-14. https://doi.org/10.1109/CGO.2019.8661197

14. Polgreen, E., Bastani, O., & Alur, R. (2020). Counterexample guided synthesis of neural network abstractions. *Proceedings of the 32nd International Conference on Computer-Aided Verification*, 176-197. https://doi.org/10.1007/978-3-030-53291-8_11

15. Prokopec, A., Rosà, A., Leopoldseder, D., Duboscq, G., Tuma, P., Studener, M., & Mosaner, T. (2019). Renaissance: Benchmarking suite for parallel applications on the JVM. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 31-47. https://doi.org/10.1145/3314221.3314637

16. Rahman, A., & Haque, M. A. (2022). Statistical challenges in performance measurement of cloud-native applications. *Journal of Systems and Software*, *193*, 111407. https://doi.org/10.1016/j.jss.2022.111407

17. Yang, B., Wang, X., Lin, C., & Zhu, T. (2025). A survey of LLM-based automated program repair: Taxonomies, design paradigms, and applications. *arXiv preprint arXiv:2506.23749*. https://doi.org/10.48550/arXiv.2506.23749

18. Yang, C., Zhao, Z., Xie, Z., Li, H., & Zhang, L. (2025). KNighter: Transforming static analysis with LLM-synthesized checkers. *arXiv preprint arXiv:2312.10012*. https://doi.org/10.48550/arXiv.2312.10012