Rethinking High-Level Code as the Primary Software Artifact
A Technical Note

Eduardo Di Santi*
College of Engineering and Applied Science, University of Colorado Boulder

April 30, 2026

Abstract

High-level programming languages have historically served as a practical compromise between human cognitive limits and executable performance. They improved readability, maintainability, and portability because humans were expected to remain the primary long-term authors and maintainers of software artifacts.

This technical note argues that reliable agentic generation and regeneration may change that optimum. If executable artifacts can be generated, repaired, translated, and re-optimized by agents, then the long-lived persistent representation of software need not remain identical to the human-facing representation. Humans may primarily control a semantic layer consisting of requirements, invariants, tests, interfaces, and optimization objectives, while agents maintain a lower-level persistent layer closer to compiler backends or hardware.

The note makes three limited contributions. First, it proposes a two-layer view of future software systems: a human semantic layer and an agent-generated persistent layer. Second, it states a minimal formal observation clarifying that conventional mediated execution cannot outperform the cost-optimal native realization on fixed hardware under a standard non-negative overhead model. Third, it offers a simple empirical illustration suggesting why bypassing repeated dependence on high-level source frontends may matter in agentic maintenance loops.

The goal is not to present a completed theory, but to mark a design direction: software ecosystems in which semantic control remains human-centered while persistent executable artifacts move downward toward machine-near representations maintained by agents.

Keywords: agentic code generation; persistent software artifacts; LLVM IR; software architecture; virtual machines; compilation

1 Introduction

Programming languages have long been understood through a familiar tradeoff: higher abstraction improves human usability, while lower-level code offers greater control over execution and machine cost [1, 4]. Historically, this tradeoff was rational because humans were expected to author, inspect, modify, and preserve the persistent software artifact over long time horizons [1].

This note asks whether that assumption should remain central once software generation and maintenance become increasingly agentic [3]. If agents can reliably generate, regenerate, repair, and optimize executable artifacts from human intent, then the representation most suitable for direct human editing need not remain the same representation that is durably stored and evolved for execution [3].

The central claim is modest but consequential: high-level languages may remain optimal as human-facing semantic interfaces while ceasing to be universally optimal as the primary software artifact.

*eduardo.disanti@colorado.edu

2 Scope and positioning

This document is a technical note rather than a full empirical or formal paper. Its purpose is to articulate a design hypothesis, clarify one structural limit of conventional mediated execution, and outline a research program around agent-generated persistent representations.

The note does not argue for manual authorship of machine-near code. Nor does it argue that abstraction should disappear. Instead, it argues that the locus of abstraction may shift upward into a semantic layer, while the durable executable layer shifts downward toward C-like, IR-like, bytecode-like, or other machine-near forms [4, 5].

3 Core thesis

The thesis of this note is that the human-facing representation of software and the persistent machine-facing representation need not coincide.

Under a human-centered regime, high-level source code served both purposes at once:

(i) it expressed semantic intent in a form humans could maintain, and

(ii) it acted as the durable source artifact from which executable behavior was repeatedly derived [1].

Under an agentic regime, these roles can be separated [3]. Humans may primarily maintain requirements, constraints, tests, interfaces, examples, and architectural intent, while agents generate and steward a lower-level persistent representation optimized for analyzability, transformation, and machine efficiency [5, 4].

4 Two-layer model

Layer 1: Human semantic layer

This layer contains human-authored intent: requirements, contracts, test suites, interface schemas, safety constraints, optimization preferences, and examples. Its purpose is semantic control, not direct execution.

Layer 2: Agent-generated persistent layer

This layer contains the durable executable representation maintained by agents. Depending on the domain, it may take the form of C-like code, LLVM IR, bytecode, WebAssembly, or other machine-near forms [4, 5]. Its purpose is not maximal human readability, but efficient execution, transformability, and verifiability [5].

5 Formal observation on mediated execution

The following observation is included only to make explicit a structural limit of conventional runtime mediation [2, 9]. It is not presented as a deep mathematical result. Its role is only to distinguish abstraction as a semantic aid for humans from mediation as a runtime cost in execution [6, 8].

Definition 5.1 (Native cost floor)**.** *For a program P on fixed hardware H, let $T_H^*(P)$ denote the infimum execution cost among all semantically equivalent native realizations of P on H.*

Definition 5.2 (Conventional mediated execution)**.** *Suppose a virtualized or mediated execution regime V executes P with cost*

$$T_V(P) = T_H(\tau_V(P)) + O_V(P),$$

where $\tau_V(P)$ is a semantically equivalent native realization and $O_V(P) \geq 0$ is the mediation overhead.

Proposition 5.1 (Simple lower bound)**.** *For every program P, one has*

$$T_V(P) \geq T_H^*(P).$$

Proof. Since $\tau_V(P)$ is a semantically equivalent native realization of P, its cost cannot be below the native cost floor $T_H^*(P)$. Adding non-negative overhead preserves the inequality. □

Remark 5.1. *This proposition is not presented as a deep mathematical result. Its role is only to distinguish abstraction as a semantic aid for humans from mediation as a runtime cost in execution.*

6 Toward a native-tight objective

The formal observation above identifies a structural ceiling for conventional mediated execution under the chosen cost model. One possible long-term objective is therefore not merely to improve the efficiency of mediated high-level execution, but to reduce the gap between realized execution cost and the native cost floor.

Definition 6.1 (Native-tight regime)**.** *A constructive execution regime M is said to be* native-tight *on a class $\mathcal{P}$ of programs if for every $P \in \mathcal{P}$,*

$$T_M(P) = T_H^*(P) + \varepsilon(P),$$

where $\varepsilon(P)$ is either zero or uniformly controlled by a small bound under the chosen cost model.

This note does not establish the existence of such a regime. It introduces the notion only as a useful target for thinking about software architectures in which abstraction is preserved at the semantic layer while the persistent executable artifact moves closer to hardware.

7 Empirical illustration

To illustrate the architectural intuition, consider a simple numerical loop implemented in two different ways: first as ordinary Python executed through the CPython stack, and second as LLVM IR constructed directly and compiled with optimization enabled.

The point of the example is not that toy loops settle architectural questions. The point is narrower: if agents can construct and regenerate lower-level representations directly, repeated dependence on high-level source frontends may cease to be architecturally central in the maintenance loop.

As a simple illustration, we compare a standard Python loop ($N = 10^7$) against LLVM IR constructed programmatically and compiled at `opt=3` [5].

This illustrative gap should not be read as a general quantitative claim about all software systems. Rather, it highlights the possibility that some costs associated with interpreted or heavily mediated execution paths need not remain architecturally central once lower-level persistent artifacts can be generated and maintained directly [2, 9, 5].

Table 1: Execution time on Apple M2 (macOS 26.3, Python 3.13).

Implementation	**Execution time (s)**	**Speedup**
Python (CPython)	0.6490	1×
LLVM IR (Agent-generated)	0.0088	≈ 74×

8 Why this may matter

The practical significance of the proposal is economic as much as conceptual. In domains such as databases, analytics engines, data pipelines, embedded systems, and infrastructure software, execution cost, latency, energy, and hardware utilization are first-order concerns.

If agents can maintain machine-near persistent artifacts under human semantic control, then part of the historical tradeoff between maintainability and execution efficiency may be renegotiated rather than merely optimized within the old architecture [4, 5].

This is especially relevant in systems such as database engines and analytics runtimes, where repeated interpretation, generic execution paths, and hardware-distance can translate directly into latency and infrastructure cost.

9 Research agenda

The proposal opens at least five questions:

1. What formal structure should the human semantic layer have?
2. Which persistent representations are best suited to agentic stewardship?
3. How should semantic alignment be verified across regeneration cycles?
4. How should portability across hardware targets be expressed?
5. In which domains does the economic gain justify this architectural shift?

These questions connect to existing work on program semantics, compilation, and code generation, but they are not settled by the present note [6, 8, 7, 3].

10 Conclusion

This technical note advances a limited architectural thesis: as software generation and maintenance become increasingly agentic, the historically natural identification between human-facing source code and the persistent executable artifact may weaken [3].

The main contribution is therefore not a theorem, but a reframing. Humans may increasingly govern semantic intent, while agents maintain lower-level persistent representations closer to compilers and hardware [5, 4]. Whether that shift becomes broadly practical remains open, but it appears important enough to name and investigate.

For that reason, the proposal may be most valuable first in domains where small architectural efficiency gains compound into large economic effects.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2 edition, 2007.

[2] Boris Feigin. Interpretational overhead in system software. Technical Report UCAM-CL-TR-797, University of Cambridge, Computer Laboratory, April 2011.

[3] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[4] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2 edition, 1988.

[5] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, Palo Alto, California, USA, March 2004.

[6] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, 1992.

[7] Clément Pit-Claudel. *Compilation Using Correct-by-Construction Program Synthesis*. PhD thesis, Massachusetts Institute of Technology, 2016.

[8] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, DAIMI, University of Aarhus, 1981.

[9] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1996.

Appendix: Repository

The code associated with the empirical illustration is available at:

https://github.com/eduardodisanti/to-boldly-go-beyond