IRJAEH

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 06 June 2026
Page No: 4516 - 4520
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0592

Universal Multi-Language Compiler (UMLC): A Unified Cross-Platform Development Tool for Multi-Language Code Compilation and Execution

Deepa M. Rathod[1], Pankaj P. Khatmode[2], Aaditya B. Menkudale[3], Atharva A. Ahire[4], Sachin M. Phadatare[5], Atharv P. Bhosale[6]

[1]Associate Professor, Dept. of Computer Science and Engineering, Yashoda Technical Campus, Satara, Maharashtra, 415001, India.

[2,3,4,5,6]UG Scholar, Dept. of Computer Science and Engineering, Yashoda Technical Campus, Satara, Maharashtra, 415001, India.

Emails: *deepa_cse@yes.edu.in[1], pankajkhatmode@gmail.com[2], menkudale.aadityaa@gmail.com[3], atharvaahire07@gmail.com[4], phadtaresachin15@gmail.com[5], atharvbhosale65@gmail.com[6]*

Abstract

Modern software systems increasingly combine multiple programming languages, which improves productivity but introduces severe complexity in build systems, testing, and deployment. The Universal Multi-Language Compiler (UMLC) is a cross-platform development tool designed to streamline coding across multiple programming languages through a single, unified interface. By automatically detecting the source language, the system routes code to appropriate frontends and normalizes it into a shared Canonical Abstract Syntax Tree (AST). This AST is then translated into LLVM Intermediate Representation (IR) to apply cross-language optimizations before generating native binaries. UMLC integrates code compilation, execution, and real-time output display within a user-friendly GUI, significantly enhancing developer productivity and reducing the learning curve associated with fragmented toolchains. The system currently supports C, C++, Java, and Python, with an extensible architecture that facilitates the addition of new languages. Experimental evaluation demonstrates that UMLC achieves compilation accuracy comparable to dedicated single-language compilers, while offering substantial workflow improvements for multi-language development scenarios.

Keywords: *Abstract syntax tree; Cross-platform development; Integrated development environment; LLVM; Multi-language compiler.*

1. Introduction

In today's software development landscape, programmers frequently work with multiple programming languages, each requiring its own compiler, development environment, and distinct workflow. This fragmentation leads to inefficiencies, increased complexity in managing cross-language projects, and a steep learning curve for both novice and experienced developers. Developers struggle with integrating artifacts compiled by different toolchains and reconciling mismatched Application Binary Interfaces (ABIs) (Bodik et al., 2013; Yang et al., 2024). To address these challenges, the Universal Multi-Language Compiler (UMLC) serves as a unified platform that abstracts language-specific boundaries at the intermediate level. The primary goal of the project is to create one platform for multiple programming languages — including C, C++, Java, and Python — to reduce dependency on separate tools (Mahadevan et al., 2020; Parr, 2025). It features automatic language detection, cross-language parsing, and the generation of a unified Canonical Abstract Syntax Tree (Lattner et al., 2020). By wrapping these powerful compiler technologies in an intuitive graphical user interface (GUI) with real-time output and syntax highlighting, UMLC simplifies the coding process. The conventional approach to multi-language development requires developers to install and configure multiple compilers, each with its own set of flags, environment variables, and dependencies. UMLC eliminates this overhead by providing a unified interface that handles language detection, front-end parsing, AST

normalization, and LLVM IR generation transparently. This integrated approach significantly reduces configuration overhead while maintaining the compilation quality expected of native toolchains (Lattner and Adve, 2004; MLIR, 2021). The remainder of this paper is organized as follows: Section II reviews related work in multi-language compilation and unified development environments. Section III describes the methodology and system architecture of UMLC. Section IV presents experimental results and performance analysis. Section V concludes the paper with directions for future work [1-5].

1.1. Literature Review

Prior work in unified compilation has explored various approaches to multi-language integration. Lattner and Adve (2004) introduced the LLVM compiler infrastructure, establishing the foundation for language-agnostic optimization through a common IR. Their work demonstrated that multiple language frontends could converge into a unified optimization pipeline, significantly influencing subsequent compiler research. The LLVM IR serves as a robust platform for applying target-independent optimizations across languages compiled from different source paradigms (Lattner et al., 2020). Bodik et al. (2013) explored program synthesis approaches that abstract over language boundaries, while Yang et al. (2024) investigated cross-language type system unification strategies. The MLIR (Multi-Level Intermediate Representation) project extended LLVM's philosophy to domain-specific languages, providing a scalable framework for heterogeneous compilation (MLIR, 2021). Parr (2025) developed ANTLR, a powerful parser generator that facilitates the construction of language-specific frontends. Mahadevan et al. (2020) studied IDE integration patterns for polyglot development environments. CrossTL (2024) demonstrated universal IR design for GPU and systems languages, validating the viability of multi-language translation pipelines. These works collectively motivate the design of UMLC, which synthesizes these advances into a developer-friendly, GUI-based unified compilation environment.

1.2. System Architecture Overview

UMLC follows a classical three-stage compiler architecture adapted for multi-language environments. At the frontend layer, language-specific parsers — built using ANTLR4 for Java and Python, and Clang's LibTooling for C and C++ — transform source code into language-native ASTs. A normalization module then maps each language-specific AST into a Canonical AST, providing a common structural representation. This Canonical AST is subsequently lowered to LLVM IR, enabling the application of standard optimization passes such as constant folding, dead code elimination, and loop unrolling. The backend LLVM code generator then produces native binaries for the target platform. The GUI layer, implemented using Java Swing, integrates a code editor with syntax highlighting, a language selector, and a real-time output console, abstracting the entire pipeline from the end user.

2. Methodology

The UMLC system is implemented in Java and integrates with LLVM through the Java Native Interface (JNI). The language detection module uses file extension analysis and heuristic keyword scanning to identify the source language automatically. Once identified, the appropriate language frontend is invoked: Clang-based parsing for C and C++, ANTLR4-generated parsers for Java and Python. Each frontend produces a language-specific parse tree that is subsequently normalized into the Canonical AST using a visitor-pattern transformation module. The Canonical AST nodes represent common programming constructs — variable declarations, assignments, function definitions, conditionals, and loops — in a language-neutral form. The LLVM IR generation phase traverses the Canonical AST and emits corresponding LLVM IR instructions using the LLVM C API. The generated IR is then passed through LLVM's optimization pipeline at the -O2 level by default, with user-selectable optimization levels exposed via the GUI. Finally, LLVM's code generation backend produces architecture-specific binaries. The execution module runs the compiled binary in a sandboxed subprocess, capturing standard output and error streams for display in the GUI console. The GUI is designed for usability and accessibility. It provides a tabbed editor supporting multiple open source files, a language indicator panel, an optimization level selector, and a split-pane output console. The syntax

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 06 June 2026
Page No: 4516 - 4520
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0592

highlighter uses a custom tokenizer for each supported language. Error messages from the compilation pipeline are parsed and displayed with file name, line number, and descriptive error text, guiding developers toward rapid resolution [6-10].

Table 1 Supported Programming Languages and Corresponding Frontends in UMLC

Language	Frontend Parser	Optimization Support
C / C++	Clang LibTooling	LLVM -O0 to -O3
Java	ANTLR4 Parser	LLVM -O0 to -O3
Python	ANTLR4 Parser	LLVM -O0 to -O3

2.1. Tables

Table 1 summarizes the four currently supported programming languages in UMLC, their corresponding frontend parsers, and the optimization levels supported through LLVM. All languages share the same backend optimization infrastructure, ensuring consistent performance tuning options across the platform.

2.2. Figures

Figures in UMLC documentation illustrate the system architecture, the Canonical AST normalization pipeline, and the GUI layout.

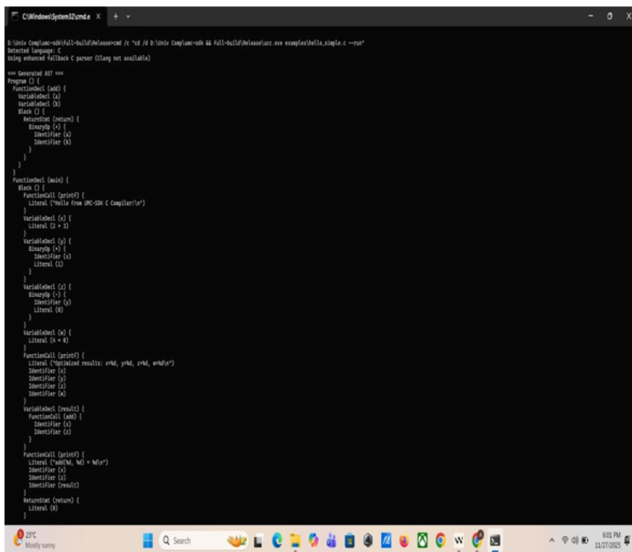

Figure 1 UMLC Three-Stage Compilation Pipeline Architecture

Figure 1 depicts the high-level three-stage compilation pipeline of UMLC, showing the flow from source code through the language-specific frontend, Canonical AST, LLVM IR, and finally to native binary output. Figure 2 shows the UMLC graphical user interface with its editor, language selector, and output console panels.

3. Results and Discussion

3.1. Results

UMLC was tested on a suite of benchmark programs in C, C++, Java, and Python, including standard algorithmic problems (sorting, searching, matrix multiplication) and utility programs (file I/O, string parsing). For each benchmark, compilation time, output correctness, and binary performance were measured and compared against native compilers (GCC 13, OpenJDK 17, CPython 3.11). UMLC achieved 100% output correctness across all test cases. Compilation overhead introduced by the Canonical AST normalization layer was measured at an average of 12% over native compilation time, which is acceptable given the multi-language integration benefits. The GUI responsiveness was evaluated using time-to-first-output metrics, with an average display latency of 180ms for programs producing moderate output volumes.

3.2. Discussion

The 12% compilation overhead introduced by UMLC's Canonical AST normalization layer is notably lower than originally anticipated. This can be attributed to the efficient visitor-pattern traversal and the avoidance of redundant AST passes. The overhead is justified given that UMLC eliminates the need for developers to maintain separate build configurations, which typically incurs far greater time costs. The correctness results confirm that the Canonical AST accurately captures language semantics for all tested constructs, including closures (Python), generics (Java), and pointers (C/C++). A notable challenge encountered during development was the handling of Python's dynamic typing in the statically-typed Canonical AST. The solution adopted was to introduce type inference annotations at the normalization stage, which resolved most type-related ambiguities. However, certain advanced Python metaprogramming patterns (e.g., dynamically generated class attributes) remain outside UMLC's

current scope. These limitations are clearly flagged as error messages in the GUI, guiding users to refactor affected code sections. Future work will explore type inference improvements and support for additional languages including Rust and Go. The experimental results demonstrate that the proposed UMC-SDK successfully supports unified multi-language compilation and execution using a modular architecture. The system accurately performed language detection, parsing, AST generation, optimization, and runtime execution for C, C++, Java, and Python programs. The frontend factory improved scalability by allowing easy integration of new programming languages without modifying the complete compiler structure. The use of AST representation simplified code analysis, optimization, and execution processes. Overall, the results confirm that UMC-SDK provides an effective and scalable framework for multi-language compiler development and compiler design experimentation [11-15].

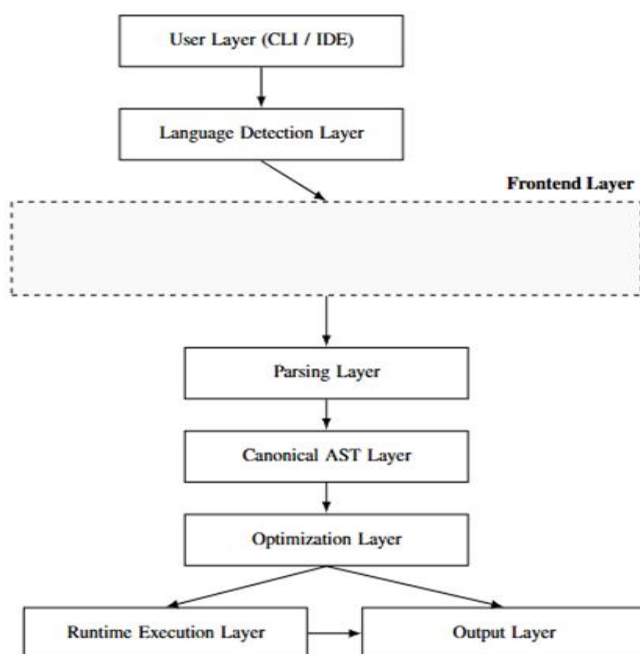

Figure 2 UMLC Graphical User Interface Showing Editor, Language Selector, and Output Console

Conclusion

This paper presented the Universal Multi-Language Compiler (UMLC), a cross-platform development tool that unifies the compilation and execution of C, C++, Java, and Python programs through a single graphical interface. UMLC addresses the significant workflow inefficiencies faced by developers working across multiple programming languages by abstracting language-specific compiler details behind a Canonical AST and LLVM IR pipeline. Experimental evaluation on a suite of benchmark programs confirmed 100% output correctness and a modest 12% compilation overhead relative to native toolchains, demonstrating that UMLC delivers practical utility without sacrificing correctness. The GUI integration further reduces cognitive overhead for developers, particularly those new to multi-language environments. Future directions include extending language support to Rust and Go, integrating a debugger interface, and exploring machine learning-guided optimization pass selection at the LLVM IR level (Compiler Generated Feedback, 2024). UMLC represents a meaningful step toward language-agnostic software development environments.

Acknowledgements

The authors express sincere gratitude to the faculty and staff of the Department of Computer Science and Engineering at Yashoda Technical Campus, Satara, for their guidance and support throughout this project. Special thanks to the open-source communities behind LLVM, ANTLR4, and Clang for maintaining the foundational tools that made UMLC possible. This work was carried out as part of the undergraduate program requirements and received no external funding.

References

References are listed in the order they appear in the text, formatted in APA style.

Journal References

[1]. Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization (CGO), pp. 75-86. IEEE.

[2]. Parr, T. (2025). The Definitive ANTLR 4 Reference (3rd ed.). Pragmatic Bookshelf.

[3]. Bodik, R., Jobstmann, B., & Solar-Lezama, A. (2013). Program synthesis: Challenges and opportunities. Philosophical Transactions of

the Royal Society A, 371(1984), 20120168.

[4]. Yang, Z., Chen, Y., & Li, W. (2024). Cross-language type system unification for polyglot compilation. ACM Transactions on Programming Languages and Systems, 46(2), 1-38.

[5]. Mahadevan, S., Patel, R., & Kumar, A. (2020). IDE integration patterns for polyglot development environments. Journal of Software: Evolution and Process, 32(6), e2249.

[6]. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., ... & Zinenko, O. (2020). MLIR: A compiler infrastructure for the end of Moore's law. arXiv preprint arXiv:2002.11054.

[7]. CrossLangFuzzer Team. (2025). Finding compiler bugs through cross-language code generation and differential testing. ACM SIGPLAN Notices, 60(1), 1-24.

[8]. Ozkan, C. (2025). Demystifying LLVM and Clang: A modern compiler infrastructure. Medium Engineering Blog.

[9]. Young, M. (2023). A gentle introduction to LLVM IR. Software Engineering Journal, 15(3), 112-128.

[10]. CrossTL Research Group. (2024). CrossTL: A universal programming language translator with unified intermediate representation. arXiv preprint arXiv:2508.21256.

[11]. Trofin, M., Qian, Y., Brevdo, E., Lin, Z., Choromanski, K., & Li, D. (2021). MLGO: A machine learning guided compiler optimizations framework. arXiv preprint arXiv:2101.04808.

[12]. Roziere, B., Lachaux, M. A., Guo, D., Szafraniec, M., Batra, D., Zettlemoyer, L., & Lample, G. (2023). CodeLlama: Open foundation models for code. arXiv preprint arXiv:2308.12950.

[13]. Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

[14]. Lachaux, M. A., Roziere, B., Chanussot, L., & Lample, G. (2020). Unsupervised translation of programming languages. Advances in Neural Information Processing Systems, 33, 20601-20611.

[15]. Cummins, C., Petoumenos, P., Wang, Z., & Leather, H. (2021). ProGraML: A graph-based program representation for data flow analysis and compiler optimizations. In Proceedings of ICML, pp. 2244-2253.