Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

Evolution of Programming Languages and Compiler Architecture: A Comparative Study of Java and Python

1st Arun Prasad
Dept. of CSE SGT University
Haryana, India

2nd Tanuj Garg
Dept. of CSE SGT University
Haryana, India

3rd Anish Brodiya
Dept. of CSE SGT University
Haryana, India

4th Rauman
Dept. of CSE SGT University
Haryana, India

Abstract*—Programming languages have progressed from hardware-dependent binary instructions and symbolic assembly mnemonics to expressive high-level languages that prioritize human readability, software modularity, and cross-platform portability. Because physical processors execute only native machine instructions, increasing linguistic abstraction has elevated the importance of compilers, virtual machines, and runtime optimization frameworks capable of bridging the semantic gap between programmer intent and hardware execution. This paper presents a literature- and specification-based comparative study of how programming-language evolution shapes compiler architecture, focusing on the reference implementations of Java (the `javac` ahead-of-time bytecode compiler and HotSpot Java Virtual Machine) and Python (the CPython compiler and virtual machine). We examine lexical and syntactic analysis, static versus dynamic type enforcement, intermediate representations, bytecode verification, adaptive interpreter specialization, and just-in-time (JIT) compilation across eleven architectural dimensions. Our analysis shows that neither Java nor Python fits a simplistic binary classification of purely compiled versus purely interpreted execution. Instead, Java combines ahead-of-time static type checking and standardized class-file bytecode generation with runtime verification and profile-guided tiered JIT compilation, whereas CPython compiles dynamic source programs into internal code objects and stack-based bytecode executed by an evaluation loop enhanced with inline caching and adaptive instruction specialization. We conclude that language semantics and compatibility goals fundamentally govern where analysis and optimization costs are placed across build time and runtime, making workload characteristics and operational constraints decisive when evaluating language-processing architectures.

***Index Terms*—Programming languages, compiler architecture, Java Virtual Machine, CPython, bytecode, just-in-time compilation, adaptive specialization.**

I. INTRODUCTION

Early digital computers required programmers to express algorithms directly in machine language, consisting of processor-specific binary operation codes and numeric memory addresses [2, 5]. Programming at the binary level demanded intimate familiarity with instruction formats and register allocation, rendering software development tedious, error-prone, and non-portable across hardware architectures. As John Backus famously characterized early low-level coding, constructing software directly for hardware resembled "hand-to-hand combat with the machine" [3]. The introduction of assembly languages replaced raw binary patterns with symbolic mnemonics such as ADD, SUB, and MOV translated by assemblers, yet assembly programs remained tightly bound to individual processor instruction sets [1].

The development of high-level programming languages in the late 1950s and 1960s—most notably FORTRAN for scientific computation, COBOL for commercial data processing, and ALGOL 60 for structured algorithmic specification—transformed software engineering by allowing developers to write problem-oriented abstractions rather than machine-oriented instructions [2, 4]. As Frederick Brooks observed within software engineering literature, the transition from machine-dependent coding to high-level programming languages improved software productivity, reliability, and simplicity by at least a factor of five [1, 5]. However, because hardware processors execute only native binary instructions, every layer of linguistic abstraction—ranging from structured control flow and recursive procedures to object-oriented polymorphism, automatic memory management, and dynamic typing—requires a corresponding language-processing system to analyze, translate, verify, and optimize source programs [1].

Java and Python represent two of the most widely deployed modern programming ecosystems, yet they embody contrasting design philosophies and implementation strategies. Java emerged in the mid-1990s with a focus on static type safety, networked portability, and managed execution through a standardized virtual machine target [12, 13]. Python, initiated in the late 1980s and released publicly in 1991, prioritized syntactic readability, rapid prototyping, and dynamic object semantics [15]. In introductory literature, Java is frequently contrasted with Python through the oversimplified dichotomy that "Java is compiled whereas Python is interpreted." While prior textbooks and surveys have examined classical compiler design [1], virtual machine optimization [10, 11], and historical language development [2, 5], there remains a need for a rigorous, side-by-side architectural synthesis that traces how Java and CPython have evolved their respective front-end parsers, intermediate representations (IRs), bytecode specifications, and runtime optimization engines through recent releases (Java SE 21 and Python 3.9–3.13).

This paper investigates how programming-language evolution drives compiler and runtime architecture through a detailed comparative study of Java (javac and the HotSpot Java Virtual Machine) and Python (explicitly focusing on the reference CPython implementation). Specifically, this study addresses five research questions:

- **RQ1:** How have programming-language abstractions evolved, and why has this evolution increased the importance of language-processing systems?
- **RQ2:** How do the Java and CPython compilation and execution pipelines differ across compile-time and runtime stages?
- **RQ3:** How do static typing, dynamic typing, intermediate representations, virtual machines, and runtime optimization

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

influence these architectures?

- **RQ4:** What practical engineering trade-offs arise from the two approaches, and why is neither approach universally superior?
- **RQ5:** Why do modern programming systems combine compilation, interpretation, runtime analysis, and optimization rather than follow a simple compiled-versus-interpreted classification?

The remainder of this paper is organized as follows. Section II reviews foundational literature on compiler construction, formal grammars, virtual machines, and adaptive optimization. Section III outlines the literature- and specification-based research methodology and comparison criteria. Section IV traces the historical evolution of programming languages and its impact on compiler complexity. Section V examines Java's language evolution and the javac/JVM compilation and execution architecture. Section VI analyzes Python's language evolution and the CPython compilation and execution pipeline. Section VII presents a structured eleven-dimension comparative analysis of Java and CPython. Section VIII discusses the principal findings regarding typing guarantees, bytecode contracts, and runtime trade-offs. Section IX states the study's limitations and identifies directions for future empirical work, and Section X concludes the paper.

II. BACKGROUND AND RELATED WORK

The theoretical and engineering foundations of language-processing systems have been established across several decades of compiler and virtual-machine research. Aho, Lam, Sethi, and Ullman [1] formalize the classical multi-phase compiler pipeline, dividing translation into a language-dependent front-end (lexical analysis, syntactic analysis, and semantic analysis), an intermediate representation (IR) and optimization middle-end, and a target-dependent back-end (instruction selection, register allocation, and code emission). The interplay among naming scopes, binding times, and type systems directly dictates the division of labor between compile-time analysis and runtime execution support [1].

Formal grammar theory underpins the front-end stages of compilation. Following the introduction of Backus-Naur Form (BNF) in the ALGOL 60 report edited by Naur [4], context-free grammars became the standard mechanism for specifying programming-language syntax and generating deterministic top-down (*LL*) and bottom-up (*LR*) parsers [1]. More recently, Ford [8] formalized Parsing Expression Grammars (PEGs), which replace ambiguous generative rules with prioritized choice operators and linear-time packrat memoization. This theoretical advance directly enabled CPython's transition from an $LL(1)$ parser to a PEG-based parser in Python 3.9 [17].

In the middle-end and back-end, intermediate representations serve as the structural bridge between source syntax and executable instructions. In production virtual machines such as the Java HotSpot Server Compiler (C2), adaptive optimizers [11] employ graph-based IRs that unify control-flow graphs (CFGs) and Static Single Assignment (SSA) data-flow dependencies, simplifying aggressive global optimizations such as speculative inlining and global value numbering. Similarly, Lattner and Adve [9] demonstrated with LLVM that a strongly typed, SSA-based IR can support lifelong program analysis, ahead-of-time (AOT) compilation, link-time optimization, and runtime just-in-time (JIT) code generation.

Extensive research has also examined bytecode virtual machines and dynamic compilation. Aycock [10] traced the historical evolution of JIT compilation from early dynamic code generation in LISP and regular-expression compilers to selective, profile-guided compilation in object-oriented virtual machines. Arnold, Fink, Grove, Hind, and Sweeney [11] surveyed adaptive optimization architectures in production virtual machines, highlighting how online profiling, inline caches, speculative devirtualization, and on-stack replacement (OSR) allow managed runtimes to overcome the startup and modular-compilation constraints of portable bytecode. Within the standard CPython reference implementation, recent architectural advances have focused on deterministic bytecode caching (PEP 552) [16], specializing adaptive interpretation (PEP 659) [18], optional removal of the Global Interpreter Lock (PEP 703) [19], and lightweight copy-and-patch JIT compilation (PEP 744) [20]. Finally, empirical and conceptual studies by Espinosa [6] and Smaragdakis [7] underscore that compiler and interpreter versions significantly affect execution efficiency and energy consumption, and that next-paradigm programming systems will increasingly rely on automated compiler reasoning and semantic guarantees.

III. RESEARCH METHODOLOGY

This study employs a qualitative, literature- and specification-based comparative technical methodology. No synthetic performance benchmarks, fabricated execution timings, or unverified empirical measurements are introduced; instead, architectural claims are grounded strictly in primary language specifications, official compiler documentation, Python Enhancement Proposals (PEPs), and peer-reviewed compiler literature.

A. Research Questions and Source-Selection Procedure

To answer RQ1 through RQ5 systematically, we conducted a targeted literature search across ACM Digital Library, IEEE Xplore, official Oracle Java SE specifications, and the official Python documentation and PEP repository. Sources were selected and verified against four inclusion criteria: (1) foundational textbooks and historical papers establishing compiler theory and language evolution [1–5]; (2) formal grammar, intermediate-representation, and next-paradigm language research [6–9]; (3) authoritative Java specifications and JVM implementation sources [11–14]; and (4) official CPython documentation, PEPs, and dynamic VM research [10, 15–20]. Every bibliographic record, DOI, and document URL was cross-checked to ensure that all 20 references are genuine and directly support the technical claims in the text.

B. Comparative Framework and Verification

To avoid conflating abstract language specifications with implementation-specific mechanisms, our comparison explicitly evaluates the standard Java SE platform (javac and Oracle/OpenJDK HotSpot JVM) alongside the reference Python implementation (CPython). The two architectures are analyzed across eleven technical dimensions: (1) original design goals, (2) typing and semantic analysis model, (3) lexical and syntactic parsing architecture, (4) intermediate representation, (5) code generation and packaging, (6) runtime execution engine, (7) optimization strategy, (8) portability model, (9) backward compatibility contract, (10) memory management and concurrency, and (11) performance-related engineering trade-offs. Version-

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

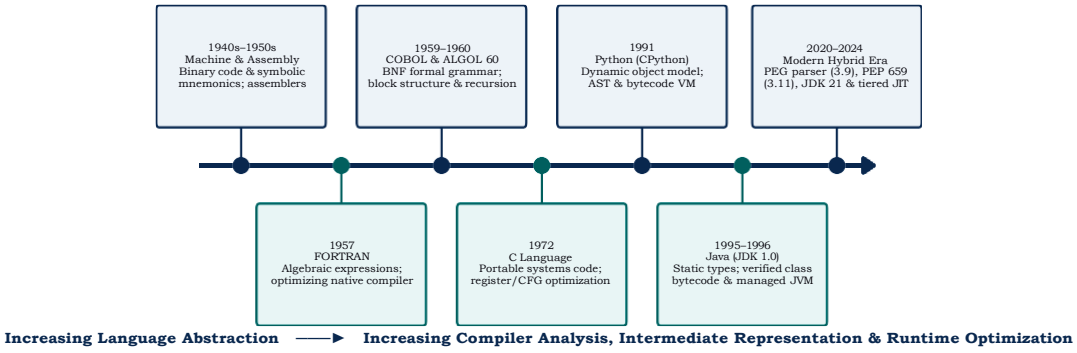


Fig. 1. Selected milestones in programming-language evolution and their influence on compiler architecture [1, 2, 4, 12, 17, 18].

dependent features—such as Java 8 invokedynamic lambdas, Java 21 virtual threads, Python 3.9 PEG parsing, Python 3.11 adaptive specialization, and Python 3.13 experimental JIT and free-threaded builds—are explicitly tied to their verified release versions. Because this methodology relies on architectural analysis rather than controlled hardware benchmarking, it establishes structural trade-offs rather than universal speed rankings.

IV. Evolution of Programming Languages

The historical evolution of programming languages reflects a continuous effort to shift cognitive complexity from human developers to automated translation and runtime systems [2, 5]. As illustrated in Fig. 1, this progression did not occur through new languages simply erasing older ones; rather, successive linguistic paradigms emerged to address expanding application domains, software scale, and hardware diversity, each imposing new demands on compiler architecture [1].

During the 1940s and early 1950s, first-generation machine languages and second-generation assembly languages required programmers to manage instruction encodings, memory layout, and control transfers manually [2, 5]. Early assemblers and linkers provided basic symbolic substitution and address relocation, but programs had to be rewritten whenever hardware changed. The emergence of third-generation high-level languages in the late 1950s marked a foundational breakthrough. FORTRAN (1957), developed by Backus and his team at IBM, proved that algebraic formulas and iteration constructs could be automatically translated by an optimizing compiler into machine code approaching the efficiency of hand-written assembly [1, 3]. Shortly thereafter, COBOL (1959) introduced English-like record and data descriptions for business processing, while ALGOL 60 (1960) established lexical block structure, nested scopes, recursive procedures, and formal syntax specification via BNF [2, 4]. These abstractions compelled compiler writers to invent formal lexical scanners, syntax-directed parsers, activation records (stack frames), and display mechanisms for non-local variable access [1].

In the 1970s, the C programming language (1972) demonstrated how high-level structured control flow and data types could be combined with low-level pointer arithmetic and explicit memory control, enabling portable operating-system and systems software development [1, 5]. Supporting C efficiently across diverse instruction set architectures catalyzed major advances in machine-independent intermediate representations, data-flow analysis, control-flow graph (CFG) transformations,

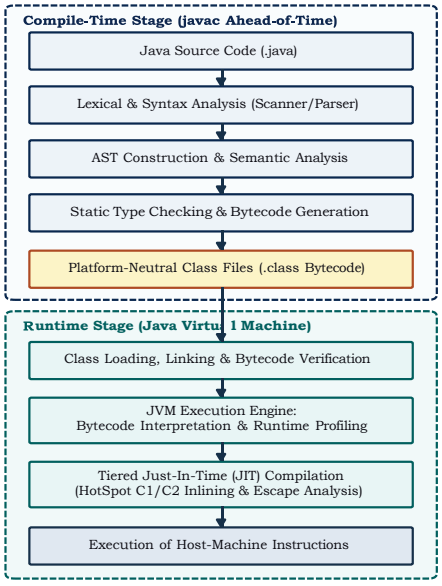


Fig. 2. Simplified Java compilation (javac) and JVM execution pipeline [11–14].

and graph-coloring register allocation [1].

As software systems grew in scale during the 1980s and 1990s, object-oriented and managed-runtime languages—exemplified by Smalltalk, C++, and Java (1995–1996)—introduced encapsulation, inheritance, dynamic dispatch (subtype polymorphism), exception handling, and automated garbage collection [5, 12]. Because virtual method calls and dynamic class loading obscure exact call targets at compile time, static whole-program optimization became insufficient, driving the rise of portable bytecode representations, bytecode verifiers, and profile-guided JIT compilers [10, 11]. Concurrently, dynamically typed scripting and general-purpose languages such as Python (1991) prioritized developer productivity, concise syntax, and runtime introspection [5, 15]. In recent years (2020–2024), the boundary between static and dynamic architectures has increasingly blurred: statically typed platforms have adopted type inference, functional streams, and lightweight virtual threads [12], while dynamic language implementations have incorporated PEG parsers, optional static type annotations, inline caching, adaptive bytecode specialization, and experimental JIT compilation [17, 18, 20].

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

V. JAVA LANGUAGE EVOLUTION AND COMPILER ARCHITECTURE

Java originated in Sun Microsystems' Green Project in 1991, where James Gosling and colleagues initially designed the Oak language for networked consumer electronics before adapting it for the World Wide Web and releasing Java 1.0 in January 1996 [5, 12]. Java's architecture was designed around object-oriented modularity, strong static type safety, memory safety via automatic garbage collection, and architecture-neutral portability summarized by the principle "Write Once, Run Anywhere" [12, 13].

A. Java Compilation and Execution Pipeline

Crucially, the standard Java compiler (javac) does not ordinarily compile Java source files (.java) directly into native processor instructions [14]. Instead, Java separates translation into a compile-time ahead-of-time bytecode compilation phase and a runtime virtual-machine execution phase, as depicted in Fig. 2.

At compile time, javac processes source code through four primary front-end stages [12, 14]:

1. **Lexical and Syntax Analysis:** The scanner converts raw character streams into tokens (keywords, identifiers, literals, and operators), and the parser constructs an Abstract Syntax Tree (AST) conforming to the formal grammar in the *Java Language Specification* (JLS) [12].
2. **Symbol Resolution and Annotation Processing:** The compiler populates symbol tables for packages, classes, interfaces, methods, and fields, and executes pluggable annotation processors (introduced in Java 6) that may generate additional source or class files [14].
3. **Semantic Analysis and Static Type Checking:** javac enforces static type conformance, access control, overload resolution, generic type-parameter bounds, and definite assignment, rejecting type errors prior to execution [12].
4. **Desugaring and Bytecode Generation:** High-level constructs are lowered (desugared)—such as performing type erasure and inserting bridge methods for generics—before emitting platform-neutral .class files containing compact stack-oriented JVM bytecode and a symbolic constant pool [13, 14].

At runtime, a compliant Java Virtual Machine (such as Oracle/OpenJDK HotSpot) executes .class files through three coordinated subsystems [11, 13]:

1. **Class Loading, Linking, and Verification:** The class-loader hierarchy dynamically loads binary class representations. During linking, the bytecode verifier performs static data-flow analysis on each method's bytecode instructions to guarantee operand-stack discipline, type-safe register access, and valid control-flow jumps before execution is permitted [13].
2. **Interpretation and Runtime Profiling:** The JVM initially executes verified bytecode via a template interpreter while collecting runtime execution profiles, including method invocation counters, loop back-edge counters, branch frequencies, and receiver-type distributions at polymorphic call sites [11].
3. **Tiered Just-In-Time (JIT) Compilation:** While different JVM implementations may use distinct execution engines, modern HotSpot implementations employ tiered compilation combining a fast client compiler (C1) with an aggressive optimizing server compiler (C2) [11]. Frequently executed "hot" code paths are compiled into native host-machine instructions using speculative class-hierarchy analysis, aggressive method inlining, scalar replacement via escape analysis, loop unrolling, and on-stack replacement (OSR), with safe deoptimization back to the interpreter if dynamic class loading invalidates a speculative assumption [11].

B. Co-Evolution of Java Features and Compiler Machinery

Table I summarizes how Java's evolution from JDK 1.0 to Java SE 21+ required coordinated enhancements across javac and the JVM runtime [12, 13]:

- **Oak / Java 1.0–1.4 (1991–2002):** Established .class bytecode, class verification, inner-class synthetic accessors (1.1), Collections, and the HotSpot C1/C2 adaptive JIT and generational garbage collectors [11, 13].
- **Java 5–7 (2004–2011):** Introduced generics via compile-time type erasure and synthetic bridge methods (preserving binary compatibility with legacy bytecode), metadata annotations, the pluggable javax.tools Compiler API (Java 6), and the JVM invokedynamic instruction (Java 7) for dynamic call-site linkage [12, 13].
- **Java 8–11 (2014–2018):** Compiled lambda expressions via invokedynamic and LambdaMetafactory rather than anonymous inner classes (Java 8); enforced strong encapsulation via module descriptors (module-info.class) in javac and class loaders alongside jshell (Java 9); and added local-variable type inference (var, Java 10) and in-memory source execution (Java 11) [12, 14].
- **Java 12–21+ (2019–2025):** Standardized switch expressions, records, sealed classes, and pattern-matching exhaustiveness analysis in javac (Java 14–21), while introducing JVM-scheduled virtual threads mounted on carrier OS threads (Java 21) and Foreign Function & Memory (FFM) interoperability [12]. Preview features in Java 22–25 require explicit –enable-preview flags [14].

VI. PYTHON LANGUAGE EVOLUTION AND COMPILER ARCHITECTURE

Python was conceived by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the late 1980s as a successor to the ABC language and was first released publicly as Python 0.9.0 in 1991 [5, 15]. Python's design philosophy emphasizes human readability through indentation-delimited block structure, a uniform object model in which all values (including functions, classes, and modules) are first-class heap objects, and dynamic typing in which variable names are untyped references bound to objects at runtime [15].

A. CPython Compilation and Execution Pipeline

A widespread misconception is that Python has no compiler and executes source text line by line. In reality, the reference implementation, CPython, incorporates a multi-stage ahead-of-execution compiler that translates Python source code (.py) into an internal intermediate representation before the virtual machine executes a single instruction [15]. At the same time,

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

TABLE I
KEY MILESTONES IN JAVA AND CPYTHON COMPILER EVOLUTION [12, 13, 16–20]

Release / Era	Language Feature	Compiler / Runtime Mechanism
Java 1.0–1.4 (1996–2002)	Classes, inner classes, Collections	javac emits .class bytecode; JVM verifier; HotSpot C1/C2 JIT [11, 13]
Java 5 (2004)	Generics, annotations, enums, autoboxing	Compile-time type erasure, synthetic bridge methods, class-file signatures [12]
Java 7–8 (2011–2014)	Lambdas, Streams, default methods	invokedynamic bytecode linkage and runtime LambdaMetafactory [13]
Java 9–17 (2017–2021)	Modules (JPMS), var, records, sealed types	module-info.class checks; flow type inference; Record class attributes [12]
Java 21+ (2023–2025)	Virtual threads, pattern matching switch	JVM continuation scheduling; pattern exhaustiveness lowering in javac [12]
Python 0.9–2.7 (1991–2010)	Indentation blocks, generators, Unicode	Tokenizer, pgen *LL*(1) parser, PyCodeObject VM, ref-counting + cyclic GC [15]
Python 3.0–3.8 (2008–2019)	Strict Unicode/bytes, async/await, type hints	Coroutine opcodes; PEP 552 deterministic hash-based .pyc caches [16]
Python 3.9–3.10 (2020–2021)	Structural pattern matching (match/case)	PEP 617 replaces *LL*(1) parser with PEG packrat parser building AST directly [17]
Python 3.11–3.12 (2022–2023)	Exception groups, adaptive interpreter	PEP 659 quickening, inline caches, in-place type-specialized opcodes [18]
Python 3.13+ (2024–2025)	Free-threaded option, experimental JIT	Optional PEP 703 no-GIL build [19]; optional PEP 744 copy-and-patch JIT [20]

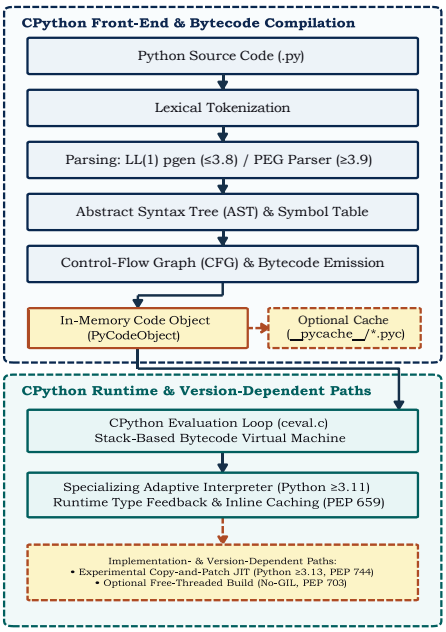


Fig. 3. Simplified CPython compilation and execution pipeline, distinguishing standard execution from optional or version-dependent paths [15–20].

one must distinguish the Python language specification from CPython: alternative implementations such as PyPy, Jython, GraalPy, Cython, and Nuitka employ different optimization and code-generation pipelines while targeting Python semantics [10, 11].

As shown in Fig. 3, the modern CPython pipeline comprises five compilation stages followed by virtual-machine execution [15]:

1. **Lexical Tokenization:** The lexer scans the input character stream, tracks indentation levels, emits explicit INDENT and DEDENT tokens to delimit syntactic blocks, and produces a stream of lexical tokens [15].
2. **Syntactic Parsing (*LL*(1) to PEG):** From Python's inception through Python 3.8, CPython relied on an *LL*(1) deterministic pushdown parser generated by pgen, which constructed a concrete parse tree (CST) before converting it into an AST [17]. Because *LL*(1) grammars permit only a single token of lookahead and disallow left recursion, natural grammar rules often required awkward refactoring or post-parsing validation. In Python 3.9 (2020), PEP 617 replaced pgen with a Parsing Expression Grammar (PEG) packrat parser based on Ford's formalism [8, 17]. The PEG parser supports infinite lookahead via memoization, handles left-recursive rules, constructs the AST directly without an intermediate CST, and enabled expressive syntax such as structural pattern matching (match/case) in Python 3.10 [17].
3. **AST Validation and Symbol-Table Analysis:** The compiler traverses the AST to build a hierarchical symbol table (symtable), statically classifying variable bindings within each lexical block as local, global (explicit or implicit), free (referenced from an enclosing scope), or cell (captured by a nested closure) [15]. Thus, while object types are resolved dynamically at runtime, lexical variable scoping in CPython is determined statically at compile time.
4. **Contr ol-Flow Graph (CFG) and Bytecode Emission:** The AST is lowered into a sequence of basic blocks forming a CFG. The compiler performs lightweight local peephole and CFG optimizations—such as constant folding, dead-code elimination after unconditional jumps, and jump-target straightening—before assembling instructions into an immutable in-memory code object (PyCodeObject) containing CPython bytecode, constants, variable names, and source-location tables [15].
5. **Optional Bytecode Caching (.pyc):** When importing modules, CPython may serialize the compiled PyCodeObject using the internal marshal format into a .pyc file inside __pycache__ [16]. Caching is strictly an optimization to bypass re-parsing and re-compiling unchanged module files on subsequent imports; top-level scripts executed directly from the command line compile to in-memory code objects without mandatory .pyc generation. Under PEP 552 (introduced in Python 3.7), .pyc invalidation can use either source timestamps or deterministic SHA-256 content hashes for reproducible builds [16]. Unlike JVM .class files, CPython bytecode is an internal implementation detail whose opcodes change across minor Python releases [15].
6. **Virtual Machine Execution and Adaptive Specialization:** The CPython virtual machine executes bytecode instructions inside the stack-based evaluation loop (_PyEval_EvalFrameDefault in ceval.c) [15]. Because standard Python opcodes such as BINARY_OP or LOAD_ATTR are generic across arbitrary runtime types, traditional CPython incurred substantial dynamic type-dispatch overhead on every iteration. To mitigate this without sacrificing dynamic semantics, Python 3.11 (2022) introduced a Specializing Adaptive Interpreter under PEP 659 [18].

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

During execution, generic "quickened" instructions monitor runtime operand types using inline caches; when a call site exhibits type stability, the interpreter rewrites the bytecode in place to a type-specialized instruction (e.g., BINARY_OP_ADD_INT or BINARY_OP_ADD_FLOAT) accompanied by lightweight type guards that deoptimize back to the generic opcode if a different object type is encountered [18].

B. *Recent Experimental Extensions in CPython*

As summarized in the lower half of Table I, Python 3.13 (released in October 2024) introduced two significant, optional implementation developments that must be carefully contextualized. First, PEP 703 introduced an experimental build configuration (`--disable-gil`) that makes the Global Interpreter Lock (GIL) optional by replacing traditional non-atomic reference counting with biased reference counting, immortal objects, and fine-grained per-object locking [19]. Second, PEP 744 introduced an experimental "copy-and-patch" JIT compiler that translates specialized Tier-2 micro-operations (uops) into native machine code using precompiled LLVM stencils [9, 20]. In standard Python 3.13 distributions, the GIL remains enabled by default and the experimental JIT is disabled by default (requiring a build-time flag or runtime configuration) [19, 20]. Therefore, these features represent evolving implementation-level capabilities rather than universal guarantees across all Python installations.

VII. Comparative Analysis of Java and Python

To synthesize the architectural relationships between Java and CPython, Fig. 4 presents a side-by-side comparison of their compilation and execution pipelines, and Table II contrasts the two systems across all eleven evaluation dimensions.

A. *Architectural Convergence and Divergence*

As illustrated in Fig. 4, both Java and CPython refute the simplistic notion that a programming language is either purely compiled or purely interpreted. Both pipelines begin with lexical tokenization, formal grammar parsing, AST construction, symbol-table resolution, and translation into a compact, stack-oriented bytecode intermediate representation [14, 15]. Furthermore, both systems execute their bytecode within a managed virtual machine that provides automatic memory reclamation, runtime exception handling, and dynamic feedback-driven optimization [13, 18].

Despite this structural parallel, the two architectures diverge fundamentally in four areas:

1. **Location of Type Resolution:** Java enforces static nominal typing during `javac` compilation [12]. Every variable, expression, and method descriptor carries an explicit compile-time type encoded directly into the `.class` file (e.g., `iadd` for 32-bit integers versus `dadd` for double-precision floats) [13]. Conversely, CPython defers object-type binding to runtime. Even when developers supply optional PEP 484 type annotations, standard CPython treats annotations as runtime metadata for external static analyzers rather than instructions for emitting statically typed bytecode [15].
2. **Role and Stability of the Bytecode IR:** In Java, the `.class` file format and JVM instruction set constitute a formal, publicly specified platform interface defined by the *Java Virtual Machine Specification* [13]. A class file compiled by an older JDK remains verifiable and executable on newer JVMs across vendors. In CPython, `PyCodeObject` and `.pyc` files are private, implementation-specific data structures [15, 16]. CPython routinely adds, removes, or renumbers bytecode instructions between minor versions (such as replacing generic binary opcodes or introducing specialized adaptive instructions in Python 3.11) to facilitate interpreter optimization [18].
3. **Verification and Security Boundaries:** Because JVM class files are designed to be distributed as binary artifacts across networks, the JVM incorporates a mandatory link-time bytecode verifier that mathematically checks type and stack safety before execution [13]. CPython assumes that bytecode is generated locally from trusted source code (or matching local `.pyc` caches) and therefore omits a formal link-time bytecode verifier, warning in official documentation that loading crafted bytecode via `marshal` can crash the interpreter [15].
4. **Optimization Depth versus Runtime Flexibility:** Because Java bytecode encodes static types and class hierarchies, HotSpot's C1 and C2 JIT compilers can perform deep interprocedural optimizations, speculative devirtualization, method inlining, and escape analysis, compiling hot methods into highly optimized native machine code [11]. In CPython, because almost any operation (such as attribute lookup or operator evaluation) can be dynamically intercepted by user-defined special methods (`__getattr__`, `__add__`), aggressive whole-method native compilation requires complex guard checks. Consequently, standard CPython prioritizes low-latency startup and in-place adaptive instruction specialization (PEP 659) [18], while delegating compute-intensive numerical kernels to native C, C++, or Rust extension libraries.

VIII. FINDINGS AND DISCUSSION

The comparative evidence assembled in Sections IV through VII directly addresses our five research questions and yields several broader insights into compiler engineering:

1) Co-evolution of Linguistic Abstraction and Compiler Complexity (RQ1): Every historical advance in programming-language expressiveness has shifted verification and optimization responsibilities onto the language-processing toolchain [1,2]. Algebraic expressions in FORTRAN required register allocation and loop optimization [3]; block structure and recursion in ALGOL 60 required formal BNF parsing and stack-frame activation records [4]; object-oriented polymorphism in Java catalyzed profile-guided JIT compilation [11]; and dynamic mutability in Python motivated inline caching and adaptive bytecode specialization [18].

2) Division of Work Between Compile Time and Runtime (RQ2 & RQ3): Java and CPython represent two coherent points on a spectrum of binding times [1]. By requiring static type declarations, Java enables `javac` to catch type mismatches ahead of time and emit typed bytecode that the JVM verifier and JIT compiler can aggressively optimize [12, 13]. However, evolving a statically typed, binary-compatible platform requires sophisticated compiler mechanisms—such as generic type erasure with synthetic bridge methods and `invokedynamic` call-site linkage—to introduce modern abstractions without breaking existing `.class` files [12]. Conversely, CPython's dynamic typing minimizes compile-time ceremony and accelerates interactive

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

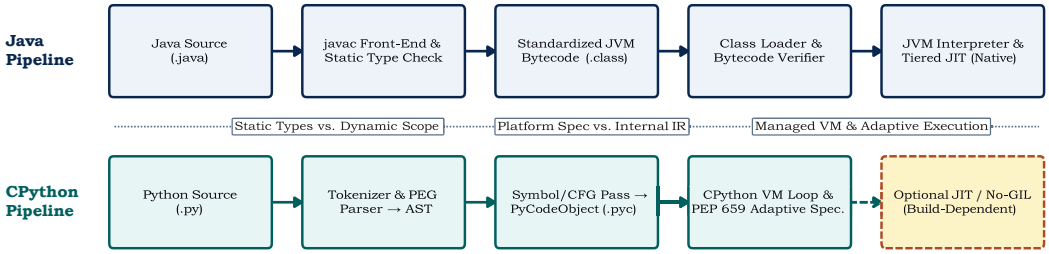


Fig. 4. Architectural comparison of Java (javac/JVM) and CPython compilation and execution pipelines [12, 13, 15, 18, 20].

TABLE II
ELEVEN-DIMENSION ARCHITECTURAL COMPARISON OF JAVA (JAVAC/HOTSPOT JVM) AND PYTHON (CPYTHON) [12, 13, 15–20]

Comparison Dimension	Java (javac & HotSpot JVM)	Python (CPython Reference Implementation)	Core Architectural Trade-Off
1. Original Design Goals	Networked portability, static type safety, sandboxing, and managed object-oriented execution [12]	Syntactic readability, developer productivity, rapid prototyping, and dynamic extensibility [15]	Upfront structural rigor and platform uniformity vs. low-ceremony expressiveness
2. Typing & Semantic Analysis	Statically and nominally typed; javac resolves types, overloads, and generic bounds before execution [12]	Dynamically and strongly typed; lexical scopes resolved statically, but object types resolved at runtime [15]	Compile-time defect detection and static type metadata vs. runtime duck-typing flexibility
3. Parsing & Syntax Processing	Deterministic scanner and parser constructing an AST per the JLS grammar [12, 14]	Indentation-aware tokenizer and PEG packrat parser (since Python 3.9, PEP 617) building AST directly [17]	Both construct ASTs; PEG parsing removed CPython's prior $LL(1)$ lookahead constraints
4. Intermediate Representation	Standardized stack-based JVM bytecode and constant pool stored in .class files [13]	AST lowered to CFG and internal stack-based bytecode inside in-memory PyCodeObject [15]	Cross-vendor specification contract (.class) vs. version-specific internal VM format
5. Code Generation & Packaging	Explicit AOT compilation step via javac producing distributable .class and JAR/module artifacts [14]	Automatic compilation on import/execution; optional .pyc caching in __pycache__ (PEP 552) [16]	Explicit build artifact generation vs. transparent compile-on-run workflow
6. Runtime Execution Engine	Bytecode verifier + template interpreter + tiered C1/C2 JIT compilers to native code [11, 13]	Bytecode evaluation loop (ceval.c) with PEP 659 adaptive specialization; optional PEP 744 JIT [18, 20]	Native code generation is central to HotSpot; specialized interpretation is central to CPython
7. Optimization Strategy	Profile-guided JIT inlining, Sea-of-Nodes IR, escape analysis, loop unrolling, OSR [11]	Compile-time constant folding; runtime quickening, inline caching, type-specialized opcodes [18]	Deep interprocedural native optimization vs. lightweight in-place bytecode specialization
8. Portability Model	Binary portability: compiled .class files execute on any compliant JVM for the target OS/CPU [13]	Source portability: .py files run across platforms, subject to interpreter version and C extensions	Binary bytecode distribution vs. source plus native-wheel environment management
9. Compatibility Contract	Strong source and binary backward compatibility via type erasure and invokedynamic [12]	Source compatibility within major series; Python 3 broke compatibility to fix core semantics; bytecode unstable	Long-term binary stability across decades vs. willingness to reform language and internal IR
10. Memory & Concurrency	Tracing garbage collectors (G1, ZGC, Shenandoah); OS threads and Java 21 virtual threads [12]	Reference counting + cyclic GC; GIL-serialized threads, asyncio, optional PEP 703 no-GIL build [19]	High-throughput parallel heap management vs. deterministic ref-counting and simple C FFI
11. Performance Trade-Offs	Higher JVM startup and JIT warm-up cost; high steady-state throughput on long-running workloads [11]	Fast edit-run turnaround and native C/C++ library offloading; higher pure-Python loop overhead [6]	Neither is universally faster; suitability depends on workload duration and native library use

development, but forces the virtual machine to perform runtime type checks and dynamic dispatch, which PEP 659 mitigates by specializing instructions dynamically using runtime type feedback [18].

3) Workload-Dependent Engineering Trade-Offs (RQ4): Our architectural analysis demonstrates why neither approach is universally superior. Tiered JIT compilation in HotSpot trades higher initial memory footprint, class-loading verification time, and profiling warm-up latency for superior steady-state native throughput on long-running, type-stable server workloads [11]. In contrast, CPython's lightweight front-end compiler and evaluation loop provide near-instantaneous startup for short-lived automation scripts, interactive data exploration, and orchestration tasks where heavy numerical arrays are processed by native C/Fortran extensions [15]. Furthermore, as Espinosa's empirical analysis of compiler and interpreter versions demonstrates, software execution time and energy consumption are sensitive not only to source language selection but also to specific runtime versions, optimization flags, and workload profiles [6].

4) Obsolescence of the Compiled-versus-Interpreted Dichotomy (RQ5): Finally, both ecosystems confirm that modern language implementations are hybrid multi-tier pipelines [10, 11]. Java combines AOT source-to-bytecode compilation (javac), runtime bytecode interpretation, and multi-tiered profile-guided JIT compilation [13]. CPython combines AOT source-to-bytecode compilation, optional deterministic bytecode caching (PEP 552) [16], bytecode interpretation, adaptive runtime instruction specialization (PEP 659) [18], and optional copy-and-patch JIT compilation (PEP 744) [20]. Similarly, alternative toolchains such as GraalVM Native Image (AOT native compilation for Java) and PyPy (meta-tracing JIT for Python [10]) illustrate that compilation and interpretation are modular engineering

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

strategies rather than fixed properties of a source language.

IX. LIMITATIONS AND FUTURE DIRECTIONS

Several limitations of this study must be stated transparently. First, this research is a qualitative, specification- and literature-based architectural study rather than an empirical benchmarking evaluation; accordingly, it does not report measured execution times, memory footprints, or energy metrics on specific hardware platforms. Second, our technical comparison centers on the standard Java SE platform (javac and HotSpot JVM) and the reference CPython implementation; alternative implementations such as Eclipse OpenJ9, GraalVM, PyPy, Cython, and MicroPython exhibit different compilation trade-offs. Third, both ecosystems continue to evolve rapidly: features such as CPython's PEP 703 free-threaded build and PEP 744 copy-and-patch JIT compiler are optional or experimental in Python 3.13–3.14 [19, 20], and preview features in Java 22–25 remain subject to refinement across releases [12].

Future work can extend this study in three directions: (1) conducting reproducible, statistically rigorous empirical benchmarks comparing compilation latency, cold-start time, warm-up curves, peak throughput, and energy consumption across HotSpot JVM, GraalVM, standard CPython 3.13, JIT-enabled CPython, and PyPy [6]; (2) investigating how static type annotations in Python (PEP 484) can be leveraged by specialized AOT and JIT compilers (such as mypyc and Cython) to approach static-language optimization efficiency; and (3) exploring next-paradigm compiler architectures that integrate formal semantic guarantees and automated algorithmic optimization, as envisioned by Smaragdakis [7].

X. CONCLUSION

This paper examined the co-evolution of programming languages and compiler architecture through a comparative study of Java and Python. Addressing RQ1, we showed that the historical progression from machine and assembly code to structured, object-oriented, and dynamic languages has continuously increased the need for automated front-end analysis, intermediate representations, and runtime optimization. Addressing RQ2 and RQ3, we demonstrated that while both Java and CPython compile source programs through lexical, syntactic, and AST stages into stack-based bytecode, Java relies on static type checking and a standardized, verifiable `.class` interface targeted at tiered JIT compilation, whereas CPython uses a PEG parser to emit internal `PyCodeObject` bytecode executed by an evaluation loop enhanced with PEP 659 adaptive specialization. Addressing RQ4 and RQ5, we established that modern language-processing systems operate as hybrid compilation-and-runtime pipelines whose design reflects fundamental trade-offs among startup latency, steady-state throughput, binary compatibility, and dynamic flexibility. Ultimately, compilers remain indispensable not merely as code translators, but as the architectural foundation that reconciles human-centric programming abstractions with efficient hardware execution.

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006, ISBN: 978-0321486813.

[2] J. E. Sammet, "Programming languages: History and future," *Communications of the ACM*, vol. 15, no. 7, pp. 601–610, Jul. 1972, doi: 10.1145/361454.361485.

[3] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978, doi: 10.1145/359576.359579.

[4] P. Naur, Ed., "Revised report on the algorithmic language ALGOL 60," *Communications of the ACM*, vol. 6, no. 1, pp. 1–17, Jan. 1963, doi: 10.1145/366193.366201.

[5] R. Kumar, C. Kaur, and S. Singh, "The evolution of programming languages: From assembly language to modern high-level languages," *Int. J. Contemp. Res. Multidiscip.*, vol. 4, no. 5, pp. 471–477, Oct. 2025, doi: 10.5281/zenodo.17481620.

[6] C. I. Olvera Espinosa, "Evolving trends in software performance and energy consumption: An analysis of compilers and interpreters over time," M.S. thesis, School of Electr. Eng., Aalto Univ., Espoo, Finland, Dec. 2023. [Online]. Available: https://aaltodoc.aalto.fi/items/42e83901-9415-4151-acd1-48dbb632557a

[7] Y. Smaragdakis, "Next-paradigm programming languages: What will they look like and what changes will they bring?" in *Proc. ACM SIGPLAN Int. Symp. New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '19)*, Athens, Greece, Oct. 2019, pp. 187–197, doi: 10.1145/3359591.3359739.

[8] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," in *Proc. 31st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '04)*, Venice, Italy, Jan. 2004, pp. 111–122, doi: 10.1145/964001.964011.

[9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation and Optimization (CGO '04)*, Palo Alto, CA, USA, Mar. 2004, pp. 75–86, doi: 10.1109/CGO.2004.1281665.

[10] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys*, vol. 35, no. 2, pp. 97–113, Jun. 2003, doi: 10.1145/857076.857077.

[11] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, "A survey of adaptive optimization in virtual machines," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 449–466, Feb. 2005, doi: 10.1109/JPROC.2004.840305.

[12] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification, Java SE 21 Edition*. Redwood City, CA, USA: Oracle America, Inc., Aug. 2023. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se21/html/index.html

[13] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, *The Java Virtual Machine Specification, Java SE 21 Edition*. Redwood City, CA, USA: Oracle America, Inc., Aug. 2023. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se21/jvms21.pdf

[14] Oracle Corporation, "The javac Command," in *Java Development Kit Version 21 Tool Specifications*, Redwood City, CA, USA: Oracle America, Inc., 2023. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/specs/man/javac.html

[15] Python Software Foundation, "Execution Model (*The Python Language Reference*) and dis — Disassembler for Python Bytecode," in *Python 3.13 Documentation*, 2024. [Online]. Available: https://docs.python.org/3/reference/executionmodel.html

[16] B. Peterson, "PEP 552 — Deterministic pycs," *Python Enhancement Proposals*, Python Software Foundation, Sep. 2017. [Online]. Available: https://peps.python.org/pep-0552/

[17] G. van Rossum, P. Galindo Salgado, and L. Nikolaou, "PEP 617 — New PEG parser for CPython," *Python Enhancement Proposals*, Python Software Foundation, Mar. 2020. [Online]. Available: https://peps.python.org/pep-0617/

[18] M. Shannon, "PEP 659 — Specializing Adaptive Interpreter," *Python Enhancement Proposals*, Python Software Foundation, Apr. 2021. [Online]. Available: https://peps.python.org/pep-0659/

[19] G. Gross, "PEP 703 — Making the Global Interpreter Lock Optional in CPython," *Python Enhancement Proposals*, Python Software Foundation, Jan. 2023. [Online]. Available: https://peps.python.org/pep-0703/

[20] B. Bucher and S. Ostrowski, "PEP 744 — JIT Compilation," *Python Enhancement Proposals*, Python Software Foundation, Apr. 2024. [Online]. Available: https://peps.python.org/pep-0744/

(This work is licensed under a Creative Commons Attribution 4.0 International License.)