Issues in Information Systems
Volume 26, Issue 2, pp. 281-291, 2025

DOI: https://doi.org/10.48009/2_iis_122

Modern programming languages – characteristics and recommendations for instruction

Pankaj Chaudhary, *NC A&T State University, pchaudhary@ncat.edu*
Lavlin Agrawal, *NC A&T State University, lagrawal@ncat.edu*
Azad Ali, *University of Fairfax, aali@ufairfax.edu*

Abstract

Academic Information Technology (IT) programs often introduce or revise courses that teach programming languages. They name these courses differently. A title frequently mentioned in these courses is "modern programming languages." Although the word "modern" may sound like the most recent, the actual aspects of what programming languages meet the definition of "modern" may need clarification. The set of requirements for classifying a programming language as "modern" may need to be defined more clearly, and instructors may ensure that all features are at least explicated in some detail, even if they are not within the scope of a particular course in a language. This paper reviews features and criteria that may need to be fulfilled by a programming language to be considered as "modern". It is expected that most languages will fulfill all these criteria. However, their strengths will vary across different features. Instructors and practitioners may need to rank order the most important criteria for selection of a language for instruction and development purposes.

Keywords: programming courses, computer programming, modern programming languages.

Introduction

The term "Modern Programming Language" is frequently encountered in academic settings, appearing in course titles, program descriptions, and curriculum updates. Related expressions include "Modern Software Development" and "Modern Computer Applications" (Ogli, 2024). In these contexts, the word "modern" typically signals contemporaneity, suggesting that the material is current and aligned with present-day practices.

The term "modern" extends beyond just indicating recency; it also signifies that the content is aligned with ongoing advancements and innovations in the field. When referring specifically to programming languages, the definition of "modern" becomes more complex. In this realm, "modern" embodies the integration of updated design principles, including enhanced readability, efficiency, safety, and support for contemporary programming paradigms, rather than merely indicating the age of the language itself. New programming languages emerge frequently, often developed to serve specific purposes. Julia is a strong example: its first stable release was launched in 2018 (Salceanu, 2018). Julia is a high-level, general-purpose dynamic programming language designed for speed and productivity, particularly in data science, artificial intelligence, machine learning, and numerical computing. It features asynchronous I/O, metaprogramming,

and efficient package management (Julia Computing, 2024). Julia prioritizes different objectives compared to other languages like Python, which focuses more on simplicity and readability, though both languages share overlapping capabilities such as metaprogramming. Each language emphasizes different strengths, and compilers and interpreters continue to evolve alongside hardware advancements like multi-core processing. Thus, no single programming language is universally superior across all feature sets (Urma et al., 2014). Consequently, the concept of a "modern" programming language is fluid: it depends less on the language's release date and more on how its features align with current computing needs and paradigms.

The purpose of this paper is to answer the last two questions: What are the features that make a programming language "modern"? The second question what is the one programming language that is considered "modern" when deciding on a language for a course name "Modern Programming Language. The paper starts by identifying the features that make programming languages "modern". Then it develops a comparison matrix where it identifies the top ranked programming languages and checks the availability of "modern" features in the selected languages.

Features of modern programming languages

While a comprehensive list of features of a modern programming language may need some discussion based on the literature review, the following features may form a common subset:

1. **Readability**: The readability of a programming language is defined by the ease with which its code can be understood (Setiawan et al., 2019). This concept is crucial in the context of team collaboration, where code authored by one individual or group may later be utilized and modified by another. The significance of readability lies in its direct impact on a program's accessibility: code that is easy to read can be readily comprehended (Bekkhus & Arvidsson, 2020; Rashed et al., 2016), whereas poorly written code presents challenges in understanding and upkeep.
   Historically, early programming languages prioritized efficiency and the streamlining of compilation processes. In contrast, modern languages are increasingly centered around readability, driven by advancements in hardware and software development capabilities. The discourse on code readability has evolved considerably, with languages like COBOL often cited as more readable due to their English-like syntax (Ali & Smith, 2014), which favors simplicity over complexity by avoiding intricate characters (e.g., {} for blocks of code or; for statement termination). This evolution reflects a broader recognition of the importance of clear, maintainable code in today's collaborative programming environments.
2. **Strongly typed**: Strongly typed languages enforce strict rules when handling data types of variables, constants, and expressions. The data type is enforced at compile time or runtime, helping to prevent unexpected results due to type mismatches (Appiah, 2021). Another important consideration is that there are no implicit type conversions that can lead to unexpected results. The language supports both static and dynamic typing, as well as type inference. The latter refers to the compiler or interpreter's ability to deduce types without explicit type annotations (Ortín, 2011).
3. **Functional Programming**: Functional Programming capabilities include first-class functions, which allow functions to be treated as variables or objects. As a result, functions can be assigned to other variables and passed as parameters to other functions, promoting code modularity and reuse (Plieskatt et al., 2014). Another critical consideration is immutability, meaning that once an object is created, its value cannot be changed. Any required modifications lead to the creation of new instances, which promotes predictability and facilitates easier debugging (Plieskatt et al., 2014). Immutability also enhances concurrency by eliminating the need for complex synchronization mechanisms, making it essential for pure functions that produce consistent outputs for the same inputs without side effects.

4. **Object-oriented Design**: Object-oriented programming (OOP) structures applications into independent objects and classes, as discussed by Ogli (2024). OOP aims to model real-world entities, facilitating code reuse in domains like software engineering and online game development. Key principles include abstraction, encapsulation, inheritance, and polymorphism. While many programming languages support OOP, some, like Java, enforce its principles more rigorously than others, such as Python. The primary advantage of object-oriented programming (OOP) is its ability to create modular software, which enhances maintainability and promotes reusability (Fallucchi & Gozzi, 2024). Additionally, encapsulation and abstraction can obscure details and sensitive information, potentially improving security
5. **Concurrency and Parallelism**: Concurrency involves dividing a program into tasks that run simultaneously through task switching, known as interleaving, even on a single processor. This improves performance, scalability, and responsiveness, as tasks, referred to as threads, can share resources and interact. While threads appear to run concurrently due to time-sharing, parallelism denotes that all threads execute independently at the same time. Programming languages implement concurrency through multithreading and asynchronous programming, allowing tasks to run in the background while other parts of the program continue executing (Yang et al., 2024). Coroutines, which enable the suspension and resumption of execution, are another mechanism used in asynchronous programming.
6. **Memory safety and Garbage Collection**: Memory safety refers to the protection against programming errors that can lead to vulnerabilities, such as buffer overflows or memory leaks, by ensuring that programs do not access memory that they are not authorized to use (Li et al., 2022). Languages like Rust, Java, and C# provide built-in memory safety features, which help prevent such issues (Wang et al., 2018). Garbage collection, a key aspect of automatic memory management, automatically reclaims memory that is no longer in use, reducing the programmer's burden and minimizing the risk of memory leaks. Through garbage collection, the system can help ensure efficient memory utilization, leading to improved application performance and stability.
7. **Interoperability**: Programming language interoperability enables different languages to interact, share data, and reuse functionality—critical for integrating legacy systems and leveraging language-specific strengths. For example, Python can control hardware on a Raspberry Pi using C/C++ drivers, facilitating modular and efficient development (Van Rossum & Drake, 2009). Interoperability supports distributed systems by allowing components in different languages to function cohesively (Henning, 2004). Mechanisms like Foreign Function Interfaces (FFI), APIs, microservices, and shared runtimes (e.g., .NET CLR) make this possible (Oracle, 2015). Notable cases include Kotlin-Java and Python-C integrations.
8. **Tooling and Ecosystem**: Tooling, especially Integrated Development Environments (IDEs) and debuggers, is crucial for optimizing code development and enhancing overall productivity. Modern programming languages increasingly benefit from AI-assisted tools such as GitHub Copilot, Codium AI, and Tabnine, which provide intelligent code completion, error detection, and contextual suggestions that streamline the coding process (Chatterjee et al., 2024). These AI tools not only aid in writing syntactically correct code but also assist in generating relevant documentation and comments, thereby improving code clarity and maintainability (Zhang et al., 2023).
   A programming language's ecosystem—including libraries, package managers, and community support—is key to extending its functionality beyond core syntax. Languages like Python, Java, Rust, and Visual Studio variants rely on ecosystems to enable code reuse, interoperability, and rapid development through community- or vendor-built packages.
9. **Modularity and Reusability**: Modularity encompasses and enables interoperability and extensions, allowing for organizing the code in separate self-contained and loosely coupled units called modules. Well-written and loosely coupled modules provide for code reusability, and if the language has interoperability features, these modules can be called from other languages. Modularity allows for code

reusability, better organization, team collaboration, easier maintainability and collaboration, and scalability (Pruijt et al., 2016). It also offers an increased level of abstraction through loose coupling and offers functionality through a well-published API.

10. **Security**: The security features in programming languages come from the amalgamation of principles like immutable data structures, strict type checking, and sandboxing. The other features, like memory-safety and garbage collection, also make languages more secure. Programming language focusing on security emphasizes on security-first principles (Khwaja et al., 2019) which include Least Privileges, Separation of Duties, Defense in Depth, Principle of Economy of Mechanism, Fail-safe Defaults, Open Design, Psychological Acceptability, Work Factor, Compromise Recording, Secure Coding Practices.
11. **Performance Optimization**: Modern programming languages are increasingly designed to optimize performance by incorporating advanced computer science principles into their compiler and interpreter architectures (Ryoo et al., 2008). These enhancements enable the generation of more efficient, faster-executing, and less resource-intensive code. Developers can further boost performance by leveraging specialized compilation flags and exploiting modern hardware features such as GPUs, leading to significantly faster execution and higher floating-point operations per second (FLOPS) (Mittal & Vetter, 2014). Most mainstream languages continue to evolve, incorporating advanced compiler optimizations to better utilize hardware resources.
12. **Cross-Platform Capabilities**: Cross-platform capabilities enable code to run across multiple hardware architectures with minimal modification. Most modern programming languages achieve this through compilers and interpreters adapted to different systems (Bekkhus & Arvidsson, 2020; Rashed et al., 2016). This often involves handling differences in CPU architectures, such as between x64-based and ARM-based Windows devices. Most prevalent languages today support multiple platforms through community or commercial efforts. Optimizing compilers and runtimes for diverse platforms is increasingly a commercial challenge rather than a technical one.

Sample of Modern Programming Languages

At present, a large number of programming languages that incorporate features characteristic of modern programming are in active use. The ranking of these languages is widely studied, with various organizations applying different criteria. Among these, the TIOBE Programming Community Index is one of the most recognized and influential benchmarks in the IT industry (Đurđev, 2024). Accordingly, the TIOBE Index was selected as the primary reference point for this analysis (TIOBE Software, 2025). The top five programming languages were considered, based on the rationale that proficiency in a widely used language enables students to develop practical skills while also gaining exposure to the core principles of modern programming. Table 1 presents the top five programming languages as of April 2025 (TIOBE Software, 2025).

Table 1. TIOBE rankings of the most popular programming languages as of April 2025

April 2025	April 2024	Programming Language	Ratings
1	1	Python	23.08%
2	3	C++	10.33%
3	2	C	9.94%
4	4	Java	9.63%
5	5	C#	4.39%

A brief description of the select features/characteristics of the programming languages selected from the TIOBE ranking is discussed below with an objective of providing general introduction to the language and lay the groundwork for comparison amongst them.

Python is a high-level, versatile programming language created by Guido van Rossum in 1991 (Van Rossum & Drake, 2003). It is valued for its readability, flexibility, and support for both object-oriented and functional programming paradigms (Szafarczyk et al., 2024). It is widely used in scripting, data science, and AI/ML thanks to its strong library ecosystem (DeVito et al., 2021). However, its parallelism and raw performance are constrained by the Global Interpreter Lock (GIL), which limits its suitability for high-performance computing tasks (Santos, 2023). Due to its interpreted nature, Python is not the fastest. However, performance can be achieved through some just-in-time and ahead-of-time compilers like Numba, mypy, Cython, and Taichi.

C++ is a powerful, multi-paradigm programming language developed by Bjarne Stroustrup in 1985 (Stroustrup, 2013). it is renowned for its high performance, fine-grained memory control, and suitability for low-level system programming (Shajarian, 2020). It is a powerful systems language offering high performance and control, though with a steep learning curve. It provides flexible abstractions and supports both object-oriented and generic programming paradigms, making it ideal for embedded systems, game engines, and real-time applications (Klepl et al., 2024). Despite these strengths, C++ can be complex and error-prone, especially in memory management and concurrent multithreaded programming (Podkopaev et al., 2016), and lacks the rapid prototyping ease of higher-level languages like Python.

C is a foundational language that powers everything from OS kernels to embedded systems. It was developed by Dennis Ritchie at Bell Labs in 1972 (Kernighan & Ritchie, 1988). It is characterized by its minimalistic syntax, which offers powerful low-level capabilities but often comes at the expense of readability and safety, especially for novice programmers. Its design emphasizes performance and efficiency, allowing direct memory access and close hardware interaction, making it a staple in systems programming (Krishnamurthi & Fisler, 2019) and embedded systems (Kandemir et al., 2004). C is statically typed, ensuring type safety at compile time, yet offers manual memory management, which, while flexible, increases the risk of errors such as buffer overflows and memory leaks (Butt et al., 2022). These characteristics make C ideal for performance-critical applications but demand rigorous discipline in software engineering practices.

C# was designed by Microsoft as an answer to Java and follows the C-like syntax. It is part of the .NET framework and was created by Anders Hejlsberg and first released in 2000 (Hejlsberg et al., 2010). C# combines high readability with modern object-oriented design, making it accessible to developers while supporting scalable, maintainable applications (Code Maze, 2023). Its static typing provides strong compile-time type checking, reducing runtime errors and enabling more robust codebases (Alomari et al., 2015). C# also features automatic garbage collection, which abstracts memory management and reduces risks like memory leaks and dangling pointers common in lower-level languages (Michaelis, 2018). Its asynchronous programming model using async/await and robust threading APIs further enhance its suitability for scalable applications. These characteristics make C# a reliable choice for enterprise and cross-platform development.

Java is a high-level programming language that follows the object-oriented paradigm in spirit and form. It was created by James Gosling at Sun Microsystems (now Oracle Corporation) in 1995 (Gosling, 2000). It is a statically typed, object-oriented programming language designed for clarity, reliability, and cross-platform portability. Its syntax is verbose but readable, supporting robust structure and maintainability in enterprise-grade applications (Varma, 2020). Java's automatic garbage collection eliminates the need for manual memory management, reducing memory-related errors and promoting safer development practices (GeeksforGeeks, 2022). The Java Virtual Machine (JVM) allows compiled Java bytecode to run on any platform, cementing Java’s reputation for “write once, run anywhere” portability (Arnold et al., 2000).

Modern Programming Language Comparison

A comparative Matrix is often used to compare characteristics when listing items side by side. In the computing field, a comparative matrix has multiple uses. For example, it can be used to compare two or three machines. While the first column lists a machine's features, the rows list the different machines being compared. As the common saying goes, "comparing apples to apples," a summary matrix can be used to compare similar products from different brands.

Next, a comparative matrix of programming language features vs. programming language is presented. This table was constructed primarily by referring to practitioner research through Google Search and other search engines. The motivation for this was that languages keep evolving, and feature discussion makes it to practitioner literature way earlier than it makes it to academic research. Table 2 below shows the matrix that we intend to fill in this study.

Table 2. Matrix for the programming languages and characteristics of modernity

Criteria	Python	C++	C	C#	Java
Readability	High – Clean, English-like syntax.	Moderate – Verbose but expressive.	Low – Minimal abstraction, terse.	High – Clear, modern syntax.	High – Verbose but readable.
Strongly Typed	Yes (Dynamic) – Types enforced at runtime.	Yes (Static) – Compile-time checking.	Yes (Static) – Strict but low-level.	Yes (Static) – Strong compile-time	Yes (Static) – Robust type system.
Functional Prog	Supported – Enhanced support (`map`, `lambda`).	Partial – Functional constructs exist.	No – Very low-level, no FP constructs.	Supported – LINQ and lambdas.	Supported – Streams and lambdas.
OOP	Yes – Supports classes and inheritance.	Yes – Full OOP with multiple paradigms.	No – Procedural only.	Full – Designed with OOP in mind.	Full – Strong OOP and interfaces.
Parallelism	GIL-limited – Multiprocessing workaround.	Strong – Threads and concurrency libraries.	Manual – Requires threads.	Strong – Native async and threading.	Strong – Threads and concurrency
Garbage Collection	Yes – Automatic memory management.	No – Manual allocation/deallocation.	No – memory managed manually.	Yes – Modern GC with tuning options.	Yes – Efficient, proven GC.
Interoperability	Strong – Works with C/C++, Java, etc.	Moderate – Can link to C libs, hard setup.	Moderate – Limited otherwise.	.NET/COM/etc. – Interop via runtime.	Strong – JNI, multiple toolkits.
Tooling & Ecosystem	Excellent – IDEs, linters, package tools.	Excellent – Mature compilers and debuggers.	Good – Stable compilers.	Excellent – Visual Studio, NuGet, etc.	Excellent – Mature and broad ecosystem.
Modular	Excellent – Modules, packages, venvs.	Excellent – Header files and libraries.	Weak – only Header inclusion	Excellent – Namespaces, assemblies.	Excellent – Packages and modules.

Criteria	Python	C++	C	C#	Java
Secure	Moderate – Relies on practices and libs.	Manual – Must handle memory and checks.	Manual – High risk of buffer overflows.	Strong – Type safety and runtime checks.	Strong – JVM handles many security concerns.
Performance	Slower – Interpreted and high-level.	Very High – Near hardware-level speed.	Very High – Fast and lightweight.	High – Compiled to efficient bytecode.	High – JIT-compiled and scalable.
Cross-Platform	Excellent – Works everywhere.	Excellent – Wide compiler support.	Excellent – Ubiquitous support.	Good – .NET Core is cross-platform.	Excellent – Works everywhere.

Conclusion and Recommendation

If one considers the matrix above, with the intention of choosing a modern programming language amongst the top five programming languages from the TIOBE April 2025 ranking (TIOBE Software, 2025), then it is clear that no single programming language may excel at everything. This is understandable. If there were one single language, then we would see the domination of that one single programming language in the application development world, and perhaps there would not be a need to do an examination like this manuscript has aimed to do.

The choice of language may boil down to the learning objective of a course and the class level. If the objective is to teach it in a class with a population of students using Windows and Mac, then one may rule out C# due to its Windows heritage. There do not seem to be any formal studies or statistics on this. Based on the authors' anecdotal experience, about 50% of the students' personal machines are Macs. While C# may be used on a Mac using Visual Studio Code, the full capabilities of the language may only be harnessed on a Windows OS. We may rule out the use of C# based on the interoperability criterion unless the use is restricted to introductory courses at a university level.

Both C++ and C use manual garbage collection. While empowering developers, manual memory management introduces significant risks of memory leaks and security risks if it is not done correctly. The languages may be used in higher-level courses where the objective is to make system-level software that is efficient and fast. Both languages shine where the software performance requirements are high. Both may also be employed to teach students the importance of memory management and writing optimized code.
Both Python and Java are good choices of languages when considering a spectrum of courses ranging from introductory to advanced. One may start with Python for good readability and ease of programming due to its interpreted nature and may switch to Java to reinforce the knowledge of Object-Oriented Concepts (Ali et al., 2023). The reason for selection is the strong object orientation of Java and the ability to map the concepts of abstraction, encapsulation, polymorphism, and inheritance directly to the language constructs (Ali et al., 2023). Both are mainstream programming languages, scoring well on all the attributes discussed in Table 2. Both may be characterized as modern programming languages.

If the objective is to give students a quick start with a reduced learning curve and basing the whole curriculum on one language, then the definitive recommendation would be Python. Python has excellent readability and is easy to start with. Students in different disciplines and those desiring to gain different levels and streams (data analytics vs. web development, etc.) of expertise can do so with its rich eco-systems

of libraries and support for all features of a modern language. Python is open source and currently is not under the control of any organization, unlike Java, which Oracle Corporation now owns. Several versions of different Java Development Kits (JDKs) exist today, including an open-source version. Due to the open-source nature of the language and the associated free IDEs like Spyder and Jupyter Lab, entry into the language is easy. While not a consideration related to a modern programming language, the available knowledge base of solutions, free training, and artificial intelligence support ensures that students have many resources available to build up their knowledge in different features of a modern programming language.

Python, as the programming language of choice, currently has much momentum. It has held the top position in the TIOBE Programming Community Index since October 2021 (TIOBE Software, 2025). It has a rich ecosystem of libraries that allows one to do tasks without writing everything from scratch, including embedded programming. It was the TIOBE's programming language of the year in 2024 (TIOBE Software, 2025). The conclusion that can be drawn here is that if teaching one programming language in a course for "Modern Programming Language", that programming language should be Python.

At the same time, what qualifies as a "modern" programming language can vary across educators, students, and industry professionals. Future research could investigate these differing perceptions through surveys or focus groups to arrive at a more user-informed understanding of modernity in programming education. Such insights could help refine curriculum design and complement the matrix-based evaluation presented in this study.

However, despite its many advantages, Python does have limitations—most notably, its interpreted nature, which makes it less suitable for performance-critical applications. Although Python can be compiled for specific platforms, languages like C++ and Java may be more appropriate for tasks that demand higher execution speed or efficient memory use, such as embedded systems. In this context, educators might use Python to introduce fundamental programming concepts due to its simplicity and readability, and then gradually transition students to more performance-oriented languages. Python, in this way, serves as a practical and accessible entry point before students move on to mastering more complex and resource-efficient programming tools.

References

Ali, A., Chaudhary, P., & Wibowo, K. (2023). Considerations for updating programming courses. *Issues in Information Systems*, *24*(2). DOI: https://doi.org/10.48009/2_iis_2023_112

Ali, A., & Smith, D. (2014). A debate over the teaching of a legacy programming language in an information technology (IT) program. *Journal of Information Technology Education: Innovations in Practice*, 13,111-127. DOI: https://doi.org/10.28945/2088

Alomari, Z., Halimi, O. E., & Sivaprasad, K. (2015). Comparative Studies of Six Programming Languages. arXiv.

Appiah, F. (2021). *Data Structures in Leelus Programming Language: A Research*. ScienceOpen Preprints.

Arnold, K., Gosling, J., Holmes, D., & Holmes, D. (2000). *The Java programming language* (Vol. 2). Reading: Addison-wesley.

Bekkhus, M., & Arvidsson, L. (2020). Resource utilization and performance: A comparative study on mobile crossplatform tools.

Butt, M. A., Ajmal, Z., Khan, Z. I., Idrees, M., & Javed, Y. (2022). An in-depth survey of bypassing buffer overflow mitigation techniques. *Applied Sciences*, 12(13), 6702.

Chatterjee, S., Liu, C. L., Rowland, G., & Hogarth, T. (2024). The impact of ai tool on engineering at anz bank an empirical study on github copilot within corporate environment. *Software Engineering*. https://doi.org/10.5121/csit.2024.140702

Code Maze. (2023, October 3). *22 C# best practices*. https://code-maze.com/csharp-best-practices/

DeVito, Z., Ansel, J., Constable, W., & Suo, M. (2021). Using Python for model inference in deep learning. arXiv. https://arxiv.org/abs/2104.00254

Đurđev, D. (2024). Popularity of programming languages. *AIDASCO Reviews, 2*(2), 24-29.

Fallucchi, F., & Gozzi, M. (2024). Puzzle Pattern, a Systematic Approach to Multiple Behavioral Inheritance Implementation in Object-Oriented Programming. *Applied Sciences*, *14*(12), 5083.

GeeksforGeeks. (2022). Java Memory Management. *GeeksforGeeks*. https://www.geeksforgeeks.org/java-memory-management/

Gosling, J. (2000). *The Java language specification*. Addison-Wesley Professional.

Hejlsberg, A., Torgersen, M., Wiltamuth, S., & Golde, P. (2010). *C# Programming language*. Addison-Wesley Professional.

Henning, M. (2004). A new approach to object-oriented middleware. IEEE Internet Computing, 8(1), 66-75.

Julia Computing. (2024). *The Julia programming language*. JuliaLang.org. https://julialang.org/ Kandemir, M., Ramanujam, J., Irwin, M. J., Vijaykrishnan, N., Kadayif, I., & Parikh, A. (2004). A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *23*(2), 243-260.

Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language*. prentice-Hall.

Khwaja, A. A., Murtaza, M., & Ahmed, H. F. (2019). A security feature framework for programming languages to minimize application layer vulnerabilities. *Security and Privacy, 3*(1). https://doi.org/10.1002/spy2.95

Klepl, J., Šmelko, A., Rozsypal, L., & Kruliš, M. (2024). *Abstractions for C++ code optimizations in parallel high-performance applications.* Parallel Computing.

Krishnamurthi, S., & Fisler, K. (2019). 13 Programming Paradigms and Beyond. *The Cambridge handbook of computing education research,* 377.

Li, Y., Tan, W., Lv, Z., Yang, S., Payer, M., Liu, Y., & Zhang, C. (2022). PACSan: Enforcing Memory Safety Based on ARM PA. *arXiv preprint arXiv:2202.03950*.

Michaelis, M. (2018). *Essential C# 7.0*. Addison-Wesley Professional.

Mittal, S., & Vetter, J. S. (2014). A survey of methods for analyzing and improving GPU energy efficiency. *ACM Computing Surveys (CSUR), 47*(2), 1-23.

Ogli, O. K. H. (2024). PYTHON AND THE EVOLUTION OF PROGRAMMING PARADIGMS: A DEEP DIVE INTO VERSATILITY. *WORLD OF SCIENCE, 7*(12), 49-55.

Oracle. (2015). The Java® Language Specification, Java SE 8 Edition. Oracle. https://docs.oracle.com/javase/specs/jls/se8/html/index.html

Ortín, F. (2011). Type inference to optimize a hybrid statically and dynamically typed language. *The Computer Journal, 54*(11), 1901-1924. https://doi.org/10.1093/comjnl/bxr067

Plieskatt, J., Rinaldi, G., Brindley, P. J., Jia, X., Potriquet, J., Bethony, J., & Mulvenna, J. (2014). Bioclojure: a functional library for the manipulation of biological sequences. *Bioinformatics*, *30*(17), 2537-2539.

Podkopaev, A., Sergey, I., & Nanevski, A. (2016). *Operational Aspects of C/C++ Concurrency*. arXiv preprint.

Pruijt, L., Wiersema, W., Werf, J. M. E. M. v. d., & Brinkkemper, S. (2016). Rule type based reasoning on architecture violations: a case study. 2016 *Qualitative Reasoning About Software Architectures* (QRASA), 1-10. https://doi.org/10.1109/qrasa.2016.7

Rashed, A., Yousif, B., & Samra, A. S. (2016). Review of FPD'S Languages, Compilers, Interpreters and Tools. Int. *Journal of Novel Research in Computer Science and Software Engineering, 3*(1), 140-158.

Ryoo, S., Rodrigues, C. I., Stone, S. S., Stratton, J. A., Ueng, S. Z., Baghsorkhi, S. S., & Hwu, W. M. W. (2008). Program optimization carving for GPU computing. *Journal of Parallel and Distributed Computing, 68*(10), 1389-1401.

Salceanu, A. (2018). *Julia Programming Projects: Learn Julia 1. x by building apps for data analysis, visualization, machine learning, and the web*. Packt Publishing Ltd.

Santos, J. (2023). Reimplementation of the SID-PSM Derivative-Free Optimization Algorithm in Python. Universidade NOVA de Lisboa. https://run.unl.pt/bitstream/10362/164325/1/Santos_2023.pdf

Setiawan, I., Maryono, D., & Basori, B. (2019). The analysis of software source code readability: Case study at education of informatics and computer engineering study program of Sebelas Maret University. *Journal of Informatics and Vocational Education, 2*(1), Article 1. https://doi.org/10.20961/joive.v2i1.35695

Shajarian, S. (2020). *The C++ Programming Language in Modern Computer Science*. Tampere University.

Stroustrup, B. (2013). *The C++ programming language*. Pearson Education.

Szafarczyk, M. Ł., Ludynia, P., & Kukla, P. Ł. (2024). *A Python library for efficient computation of molecular fingerprints*. arXiv. https://arxiv.org/abs/2403.19718

TIOBE Software. (2025, April). *TIOBE index for April 2025.* Retrieved March 4, 2025, from https://www.tiobe.com/tiobe-index/

Urma, R., Orchard, D., & Mycroft, A. (2014). Programming language evolution workshop report. *Proceedings of the 1st Workshop on Programming Language Evolution*. https://doi.org/10.1145/2717124.2717125

Van Rossum, G., & Drake, F. L. (2003). Python language reference manual.

Van Rossum, G., & Drake, F. L. (2009). PYTHON 2.6 reference manual.

Varma, S. C. G. (2020). The Role of Java in Modern Software Development: A Comparative Analysis with Emerging Programming Languages. *International Journal of Emerging Research in Engineering & Technology.*

Wang, F., Song, F., Zhang, M., Zhu, X., & Zhang, J. (2018, August). Krust: A formal executable semantics of rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)* (pp. 44-51). IEEE.

Yang, Z., Kung, F. Y. H., & Schneider, D. W. (2024). Individual preferences in multiple goal pursuit: reconsidering the conceptualization and dimensionality of polychronicity. *Applied Psychology, 74*(1). https://doi.org/10.1111/apps.12575

Zhang, B., Liang, P., Zhou, X., Ahmad, A., & Waseem, M. (2023). *Practices and challenges of using github copilot: an empirical study.* https://doi.org/10.48550/arxiv.2303.08733