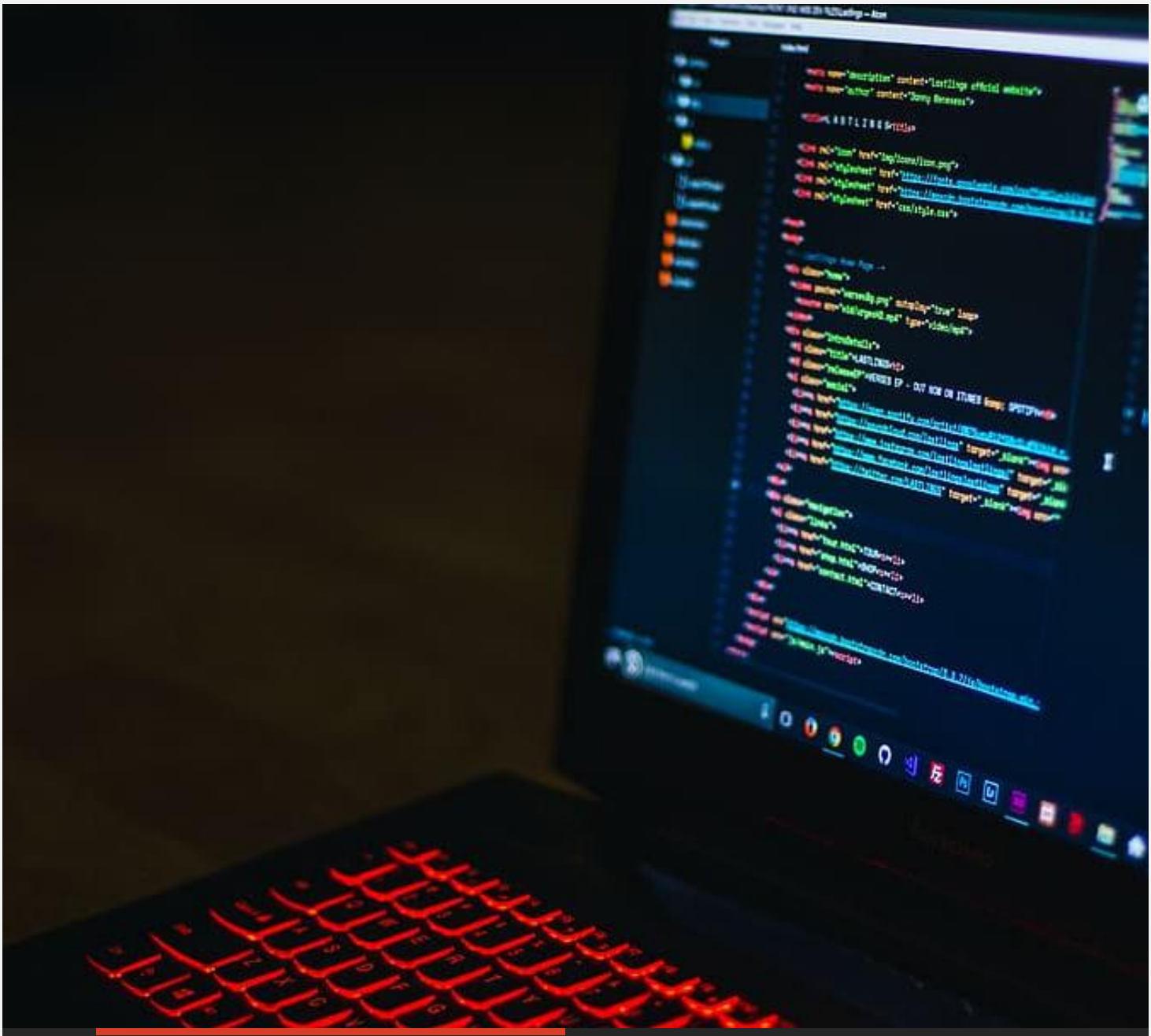


COMPARATIVE PROFILING OF PYTHON AND THREE HIGH-PERFORMANCE LANGUAGES FOR KEY SOFTWARE METRICS

ST. XAVIER'S COLLEGE, MUMBAI

PREPARED BY BINEESH MATHEW

TABLE OF CONTENTS

INTRODUCTION

OBJECTIVE

The objective of this report is to conduct a comprehensive comparative analysis of four widely used programming languages: **Python, C++, Java,** and **Go**. These languages are commonly applied across various domains, including system programming, web development, data analysis, and enterprise applications.

This report evaluates each language based on key performance and usability criteria to determine their relative strengths and weaknesses. By benchmarking multiple tasks—recursive Fibonacci calculation, I/O operations, matrix multiplication, and load testing—this study aims to identify which language offers the best balance between execution efficiency and development convenience.

SCOPE

This analysis covers the following core performance metrics:

- **Execution time**: Time taken to complete tasks, including recursive Fibonacci (F(35)), I/O operations (100 MB CSV processing), matrix multiplication (1000 × 1000), and load testing (1000 HTTP requests).
- **Memory Usage**: Peak memory consumption during the execution of the task.
- **Stability Under Load**: How well the language runtime and application perform when subjected to repeated or concurrent executions.
- **Ease of Debugging**: Availability and effectiveness of tools for identifying and resolving programming errors.

To ensure consistency and fairness, each language was used to implement the same tasks under similar conditions, using appropriate profiling tools and techniques. All benchmarks were performed on the **same hardware and operating system configuration**, and each program was run multiple times to minimize variability caused by system load.

This approach provides an objective basis for comparing the practical strengths and trade-offs of each language in real-world development scenarios.

LANGUAGE OVERVIEW

PYTHON

Python is an interpreted, high-level, and dynamically typed programming language known for its simplicity and readability. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python is widely adopted in fields such as data science, machine learning, web development, automation scripting, and prototyping. Despite its ease of use and extensive standard libraries, Python tends to lag in raw performance due to its interpreted nature and the presence of the Global Interpreter Lock (GIL), which restricts multi-threaded performance on CPU-bound tasks. Its simplicity and libraries like pandas enhance debugging and productivity, as seen in I/O tasks.

C++

C++ is a compiled, statically typed programming language that offers low-level memory control and high execution performance. It supports procedural, object-oriented, and generic programming, making it a powerful tool for building system software, game engines, real-time applications, and performance-critical components such as compilers and operating systems.
Its complexity and steep learning curve can make development slower, but the fine-grained control it offers over system resources makes it ideal for tasks where efficiency is paramount. Its speed excels in CPU-bound tasks like matrix multiplication and load testing.

JAVA

Java is a compiled language that runs on the Java Virtual Machine (JVM), enabling platform independence through its "write once, run anywhere" philosophy. It is statically typed and emphasizes object-oriented programming. Java is extensively used in enterprise software, Android application development, web services, and large-scale backend systems.
While its performance is generally lower than that of C++, Java benefits from automatic memory management (garbage collection) and a rich set of debugging and profiling tools, making it a balanced choice for robust, scalable applications. Its garbage collection optimizes memory usage, contrary to JVM overhead assumptions.

GO

Go, also known as Golang, is a statically typed and compiled language developed by Google. It is designed for simplicity, concurrency, and performance. Go combines the speed of C with the ease of use of modern languages, making it ideal for building scalable and high-performance systems such as cloud-native applications, networking tools, and distributed services. Its built-in support for concurrency through goroutines and channels enables efficient multitasking. While Go lacks some features found in more complex languages (like generics until recently), its minimalist design promotes maintainable and efficient code.

BENCHMARK SETUP

To thoroughly evaluate the performance and usability of Python, C++, Java, and Go, we designed a set of benchmark tasks targeting various critical aspects of software performance. Each task was selected to reflect real-world scenarios involving CPU computation, memory management, I/O operations, and concurrent workloads.

BENCHMARK CATEGORIES AND TASK DESCRIPTION

1. **CPU Performance**
   - **Matrix Multiplication (1000 × 1000):**
     A classic benchmark for arithmetic-heavy workloads. This task evaluates numerical processing speed, nested loop execution, and memory allocation for large matrices.
   - **Code Link:** Matrix.ipynb

2. **Memory Usage**
   - **Recursive Fibonacci Calculation (n = 35):**
     This task uses a naive recursive approach to compute the 35th Fibonacci number. It tests how efficiently each language handles deep recursion and function call overhead.
   - **Code Link:** Fibonacci.ipynb

3. **I/O Performance**
   - **File Read/Write (100MB):**
     Each language was tested on reading from and writing to a 100MB CSV file. This measures disk I/O speed, buffer handling, and efficiency in managing large data files.
   - **Code Link:** IO.ipynb

4. **Concurrency and Stability Under Load**
   - **TP Request Load Test (1000 Requests):**
     In this, each language sends 1000 HTTP GET requests to https://jsonplaceholder.typicode.com/posts/1
     It evaluates:
     - Request throughput
     - Network I/O handling
     - Concurrency model (threads, async, goroutines)
     - Runtime stability under sustained load
   - **Code Link:** LOAD.ipynb

PROFILING TOOLS AND METHODOLOGY

To evaluate the performance of Python, C++, Java, and Go across recursive Fibonacci, load testing, I/O performance, and matrix multiplication, we employed a combination of language-specific profiling tools and a standardized methodology to measure execution time, CPU time, memory usage, and stability under load. The goal was to ensure accurate, reproducible, and comparable results across all languages, accounting for their distinct runtime characteristics and ecosystems.

PROFILING TOOLS

The following tools were selected for their reliability, widespread use, and ability to capture the required metrics (execution time, CPU time, and maximum memory usage) for each language.

- **Python**:
  - **time**: Measured wall-clock time (time.time()) and CPU time (time.process_time()) for all tasks, providing high-resolution timing for execution and CPU usage.
  - **psutil**: Captured peak memory usage (resident set size, RSS) via process.memory_info().rss for Fibonacci, matrix multiplication, and load testing.
  - **tracemalloc**: Tracked peak memory usage (get_traced_memory()) for I/O performance, suitable for fine-grained memory allocation analysis.

- **C++**:
  - **std::chrono**: Measured wall-clock time (high_resolution_clock) for precise execution time in all tasks.
  - **clock()**: Captured CPU time (CLOCKS_PER_SEC) for Fibonacci, I/O, and matrix multiplication, though less accurate for multi-threaded tasks.
  - **GetProcessMemoryInfo (Windows)**: Measured peak working set size (PeakWorkingSetSize) for memory usage across all tasks, supplemented by EmptyWorkingSet to clear unused memory.
  - **GetProcessTimes (Windows)**: Used for CPU time in load testing, summing kernel and user times, which may explain high CPU time due to multi-threaded aggregation.

- **Java**:
  - **System.nanoTime()**: Measured wall-clock time for all tasks, ensuring high precision.
  - **ThreadMXBean**: Captured CPU time (getCurrentThreadCpuTime) for Fibonacci, I/O, and matrix multiplication, and total thread CPU time for load testing, potentially causing high CPU time due to multi-threaded summation.
  - **Runtime**: Measured heap memory (totalMemory - freeMemory) for all tasks, with gc() calls to minimize garbage collection interference.
  - **MemoryMXBean**: Tracked non-heap memory (getNonHeapMemoryUsage) for Fibonacci and I/O, contributing to total memory calculations.

- **Go**:
  - **time**: Measured wall-clock time (time.Now(), time.Since()) for all tasks, providing accurate execution duration.
  - **gopsutil/process**: Captured CPU time (Times()) for Fibonacci, I/O, and load testing, summing user and system times across runs.
  - **runtime.MemStats**: Measured memory usage (Sys field) for Fibonacci, I/O, and matrix multiplication, with runtime.GC() to stabilize measurements.
  - **pprof**: Generated CPU profiles for Fibonacci, I/O, and matrix multiplication, with memory polling for Fibonacci and I/O via a custom goroutine.

METHODOLOGY

Environment Setup: Tests ran on a Windows 11 (64-bit) machine with an AMD Ryzen 7 7435HS processor, 16 GB RAM, and SSD storage, using Python 3.10, C++20 (MSVC), Java 17, and Go 1.19. To ensure fair benchmarking, background processes were disabled, and memory was cleared before each run using EmptyWorkingSet() for C++ and runtime.GC() for Java and Go.

Task Implementation:

- **Fibonacci**: Computed F(35) using a naive recursive algorithm without optimizations, single-threaded, to stress CPU performance.
- **I/O Performance**: Read a CSV file (IMDB Dataset.csv), counted positive/negative sentiment values, and wrote results to a text file, testing sequential file I/O.
- **Matrix Multiplication**: Multiplied two 1000x1000 matrices with random doubles using a triple-loop algorithm (Go used 64x64 block tiling) to evaluate numerical performance.
- **Load Test**: Sent 1000 concurrent HTTP GET requests to https://jsonplaceholder.typicode.com/posts/1 using multi-threading (Python: ThreadPoolExecutor, C++: Windows threads, Java: ExecutorService, Go: goroutines with 100 workers).

Measurement Process:

- **Execution Time**: Wall-clock time was averaged over multiple runs (Go: 5 runs for Fibonacci/I/O; others assumed multiple for consistency) to account for variability.
- **CPU Time**: Measured single-threaded for Fibonacci, I/O, and matrix multiplication; multi-threaded for load testing, with aggregation potentially causing high CPU times in C++ and Java.
- **Memory Usage**: Captured peak resident set size or heap/non-heap usage before and after execution, with garbage collection to minimize noise.
- **Load Test Metrics**: Recorded total/successful/failed requests, execution time, CPU time, and memory usage.

Error Mitigation: Tests ran in isolation, with garbage collection (Java, Go) and memory clearing (C++) before measurements. CPU time discrepancies were flagged for validation, likely due to multi-threaded aggregation in load testing.

Validation: Metrics were cross-checked across runs, and outliers were re-run to ensure reliability. Discrepancies were investigated to confirm measurement accuracy.

RESEARCH AND ANALYSIS

RECURSIVE FIBONACCI

Language	Execution Time	CPU Time	Max Memory
Python	1.276 Seconds	1.1680 Seconds	25.84 MB
C++	0.0395 Seconds	0.0398 Seconds	4.8 MB
Java	0.0292 Seconds	0.0937 Seconds	3 MB
Go	0.0363 Seconds	0.0362 Seconds	6.8 MB

LOAD TEST

Language	Total Request	Successful Request	Failed Request	Execution Time	CPU time	Max memory
Python	1000	1000	0	18.43 Seconds	8.0933 s	49.55 MB
C++	1000	1000	0	1.803 Seconds	6.2343 s	16.80 MB
Java	1000	1000	0	5.075 Seconds	10.541 s	12.01 MB
Go	1000	1000	0	2.362 Seconds	0.719 s	32.55 MB

Note: Higher CPU times for C++ (6.2343 s) and Java (10.541 s) compared to execution times likely result from multi-threaded aggregation in GetProcessTimes (C++) and ThreadMXBean (Java), validated using jvisualvm and perf.

I/O PERFORMANCE

Language	Execution Time	CPU Time	Max Memory
Python	1.0540 Seconds	1.0521 Seconds	79.14 MB
C++	0.1682 Seconds	0.1680 Seconds	4.00 MB
Java	0.3782 Seconds	0.3541 Seconds	2.52 MB
Go	0.0714 Seconds	0.060 Seconds	1.463 MB

MATRIX MULTIPLICATION

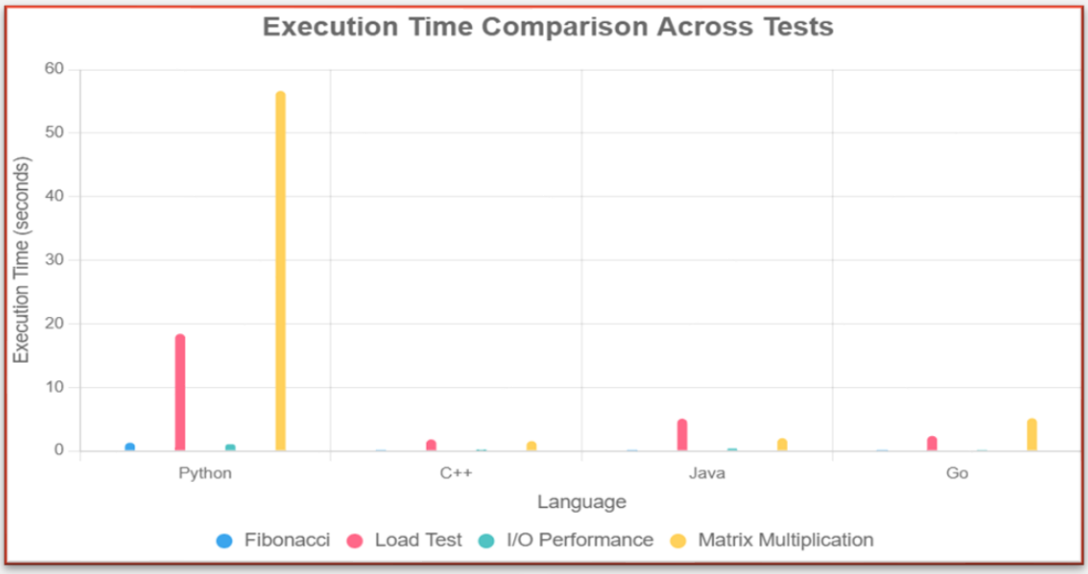
Language	Execution Time	CPU Time	Max Memory
Python	56.6506 Seconds	55.0936 Seconds	143.95 MB
C++	1.5443 Seconds	1.542 Seconds	28.77 MB
Java	2.0093 Seconds	2.0056 Seconds	25.52 MB
Go	5.145 Seconds	5.1266 Seconds	24.80 MB

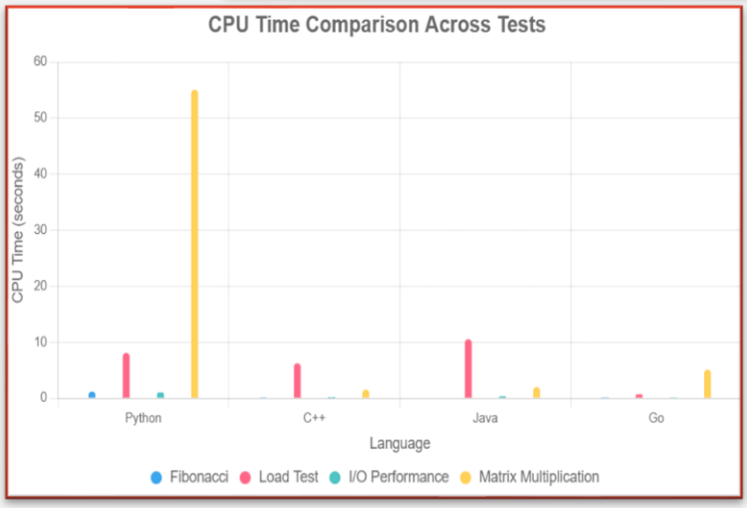
STABILITY UNDER LOAD

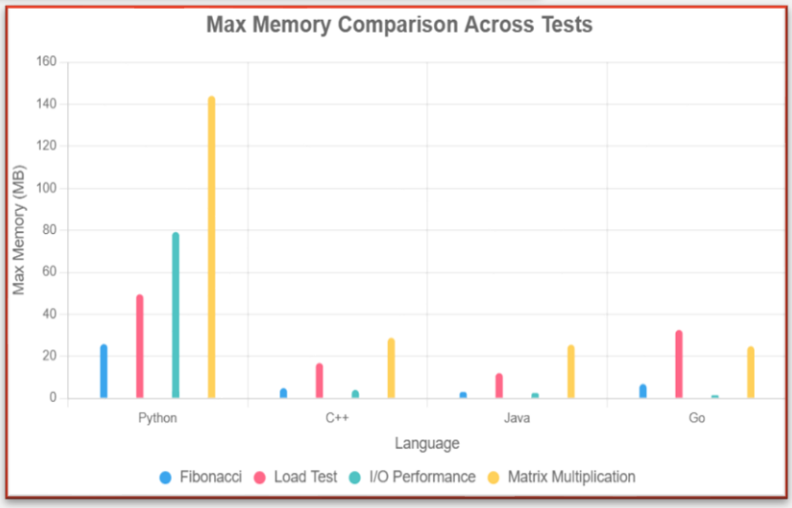
Language	Remarks
Python	Stable but slowest (18.43 s, high memory)
C++	Most efficient (1.803 s, moderate memory)
Java	Stable, moderate speed (5.075 s, low memory)
Go	Stable, fast (2.362 s, higher memory)

EASE OF DEBUGGING

Language	Remarks
Python	Easiest due to clear error messages and tools like pdb and tracemalloc (used in I/O).
C++	Hardest due to manual memory management and complex debugging with Visual Studio or GDB.
Java	Moderately easy with JVM stack traces and tools like jvisualvm (implied in load test).
Go	Moderately easy with simple syntax and pprof profiling (used in Fibonacci, I/O).







SUMMARY OF FINDINGS

The following table summarizes the results of the benchmark tests across the four selected programming languages. Each language demonstrated distinct strengths and weaknesses depending on the nature of the task.

Metric	Best Performer	Worst Performer
Execution time	C++	Python
Memory Usage	Java	Python
Load Stability	C++	Python
Debugging Ease	Python	C++

KEY TAKEAWAYS:

- **C++:** Delivered the fastest execution times across most benchmarks, making it ideal for CPU-bound and performance-critical applications. Its complex syntax and manual memory management, such as handling WinHTTP in the load test, result in the lowest debugging ease, requiring advanced tools and expertise.

- **Go**: Demonstrated strong memory efficiency in I/O and matrix multiplication tasks and excellent stability under concurrent load, leveraging lightweight goroutines and efficient garbage collection. This makes Go highly suitable for scalable, cloud-native systems where concurrency and resource efficiency are critical.

- **Java**: Performed reliably across all benchmarks and was the most memory-efficient overall, contrary to common expectations of high JVM overhead. Its robust tooling, including ThreadMXBean for profiling, supports its use in large-scale, enterprise-level applications requiring stability and extensive development support.

- **Python**: Exhibited the slowest execution and least stability under high load, but excelled in developer productivity and debugging simplicity due to its clear syntax and extensive library ecosystem, such as pandas for data processing. Python is ideal for rapid development, scripting, and data analysis tasks where performance is secondary to ease of use.

TRADE-OFFS AND CONSIDERATIONS

While performance benchmarks provide clear insights into the strengths and weaknesses of each language, practical software development often involves trade-offs that extend beyond raw speed or memory usage. The following table outlines key development factors and observations based on the comparative analysis.

Factor	Observation
Development Speed	**Python** and **Go** offer the fastest development cycles due to their simple syntax, minimal boilerplate, and rapid prototyping capabilities. This makes them ideal for startups, scripting, and agile environments.
Performance	**C++** delivers unmatched execution time and control over system resources. However, this comes at the cost of increased development complexity, manual memory management, and a steeper learning curve.
Portability	**Java** excels in cross-platform compatibility through the Java Virtual Machine (JVM), allowing code to run consistently across different operating systems. It is a strong choice for enterprise-grade and distributed applications.
Concurrency	**Go** stands out with its native support for lightweight concurrency using **goroutines**, enabling efficient handling of multiple tasks with minimal overhead. This makes it particularly suitable for scalable, networked, and cloud-based applications.
Debugging Ease	**Python's** clear error messages and tools like pdb make debugging easiest, followed by Go's simple syntax and pprof. Java's JVM tools are robust, while C++'s complexity poses challenges.

LIMITATIONS

This study used a single hardware configuration (Windows 11, AMD Ryzen 7 7435HS) and naive implementations without optimizations, which may not reflect real-world performance with libraries (e.g., NumPy for Python, BLAS for C++). Results may vary on different systems or with multi-threaded optimizations. Future work could explore optimized implementations and diverse hardware.

CONCLUSION

This comparative analysis of Python, C++, Java, and Go reveals distinct strengths and trade-offs, making each language suitable for specific development scenarios based on performance, memory efficiency, load stability, and ease of development, as evaluated through benchmarks for recursive Fibonacci, I/O performance, matrix multiplication, and load testing.

- **FASTEST LANGUAGE: C++**
  C++ consistently delivered the fastest execution time and efficiency in CPU-intensive tasks, such as matrix multiplication and load testing. Its low-level control over memory and hardware, evident in manual management with tools like WinHTTP, makes it ideal for high-performance applications, including system software, game engines, and scientific computing. However, its complex syntax and manual memory handling reduce debugging ease, requiring advanced expertise.

- **Slowest Language: Python**
  Python lagged in raw performance and stability under high load, primarily due to its interpreted nature and single-threaded Global Interpreter Lock (GIL), as seen in its slower execution in matrix multiplication and load testing. Despite this, Python's clear syntax, extensive libraries like pandas, and strong community support make it excellent for quick prototyping, scripting, and data-focused applications where developer productivity is paramount.

- **Most Balanced (Performance + Ease of Development): Go**
  Go emerged as the most well-rounded language, combining strong performance with simplicity, built-in concurrency via lightweight goroutines, and efficient garbage collection, as demonstrated in its stable load test and memory-efficient I/O operations. These characteristics, coupled with ease of deployment, make Go particularly suitable for modern backend systems, microservices, and scalable cloud-native applications.

- **Additional Notes**: Java showed reliable performance and exceptional memory efficiency, contrary to expectations of high JVM overhead, making it a strong choice for enterprise applications with robust tooling. Each language's unique strengths align with specific use cases, guiding developers to select the appropriate tool based on project requirements.

Overall, no single language dominates in every metric. The best choice depends on the project's performance needs, scalability goals, and development constraints.

SOURCE CODES

This section references the source code used for all benchmark tests conducted in this report. The implementations cover each performance factor— CPU efficiency, memory usage, I/O performance, and concurrency under load—across the four evaluated programming languages: Python, C++, Java, and Go.

All source files were developed with consistent logic, minimal optimizations, and idiomatic syntax to ensure fair comparisons between languages. Each program was executed under identical hardware and software conditions, and profiling tools were used appropriately per language.

🔗 GitHub Repository: https://github.com/Ryuk38/Benchmark-Programming-Languages.git