# OpenCV Performance Evaluation

**Brian Lam, Don Huynh, Jorge Luis Suarez, Benjamin Hong**

**{lam, donhuynh, bjhong, jorgeluiss}@cpp.edu**

## Introduction

Our research focuses on benchmarking the execution speed, CPU and memory usage, and power consumption of five specific OpenCV image processing functions using four different languages on the Raspberry Pi 4, Model B. The five functions are image cropping, image resizing, image sharpening, blue to gray color conversion, and histograms. Each of these OpenCV functions included four different programs to perform the image processing – A C++ program, a Python program, a Golang program, and a Rust program. C++'s low-level optimizations, Python's high-level scripting, Golang's concurrency support, and Rust's memory safety provides us with four solid language platforms for each of the test functions on our testbed, the RPi.

## Aim

This study aims to comprehensively benchmark and analyze the performance metrics of five distinct OpenCV image processing functions—image cropping, resizing, sharpening, color conversion from blue to gray, and histogram generation—across four diverse programming languages on the Raspberry Pi 4, Model B. The selected languages for evaluation encompass C++, Python, Golang, and Rust, each offering unique attributes in terms of optimization, scripting, concurrency support, and memory safety. Through this analysis, the study aims to ascertain the suitability of Rust, renowned for its memory safety and performance, especially in scenarios necessitating robust security, system-level programming, and high-performance applications. Additionally, it seeks to evaluate Golang's strengths in concurrent systems, scalable network servers, and cloud-based applications, while considering whether adopting new implementations, like Rust or Golang, offers substantial benefits over established languages like C++ and Python for OpenCV image processing tasks.

## Method

**Execution Time Measurement**:

Utilized OpenCV's library function to calculate execution time following documented procedures.

```
e1 = cv.getTickCount()
# your code execution
e2 = cv.getTickCount()
time = (e2 - e1)/ cv.getTickFrequency()
```

*Figure 1: OpenCV Execution Time Calculation [1]*

**Automated Benchmarking Process**:

Developed a bash script to automate benchmarking procedures, including:

- Deletion of existing output files
- Execution of each script
- Capture of execution times for evaluation

**Testing Methodologies**:

Single Run Method:

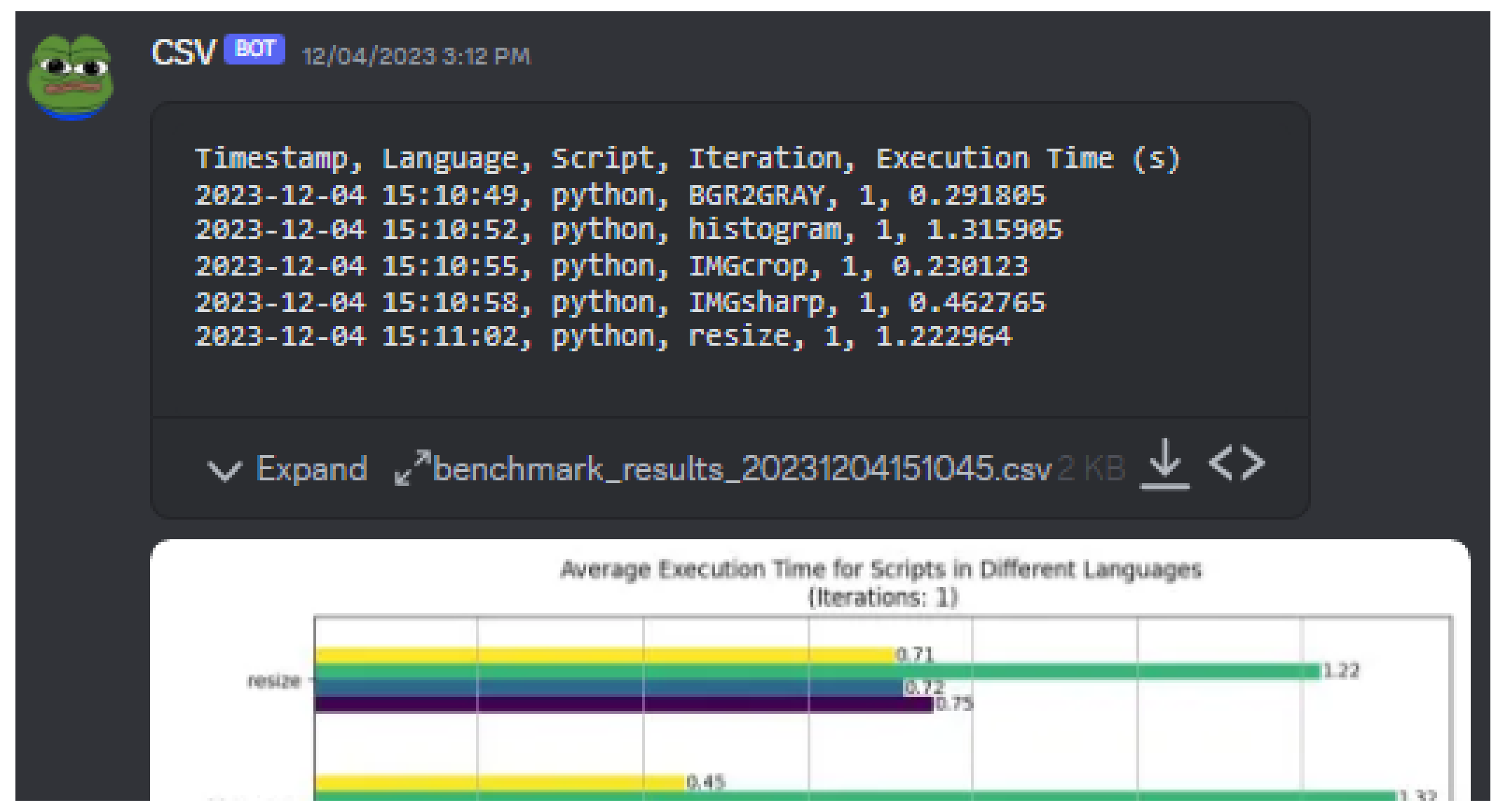
- Executed all 20 scripts once
- Recorded individual execution times
- Organized data into a CSV file for analysis

Iterative Method:

- Enabled user-defined iterations for each script
- Captured execution times across defined iterations
- Derived stable performance metrics by averaging times

**Result Dissemination**:

- Automated generation of a chart summarizing collected data using Python
- Transmission of both CSV file and chart image via Discord for convenient access and visualization of benchmarking outcomes


CSV
BOT
12/04/2023 3:12 PM
Expand
benchmark_results_20231204151045.csv 2 KB
Average Execution Time for Scripts in Different Languages
(Iterations: 1)
resize
0.71
1.22
0.72
0.75
0.45

```
Timestamp, Language, Script, Iteration, Execution Time (s)
2023-12-04 15:10:49, python, BGR2GRAY, 1, 0.291805
2023-12-04 15:10:52, python, histogram, 1, 1.315905
2023-12-04 15:10:55, python, IMGcrop, 1, 0.230123
2023-12-04 15:10:58, python, IMGsharp, 1, 0.462765
2023-12-04 15:11:02, python, resize, 1, 1.222964
```

*Figure 2: Benchmark Result Output via Discord*

## Results

**Single Run**

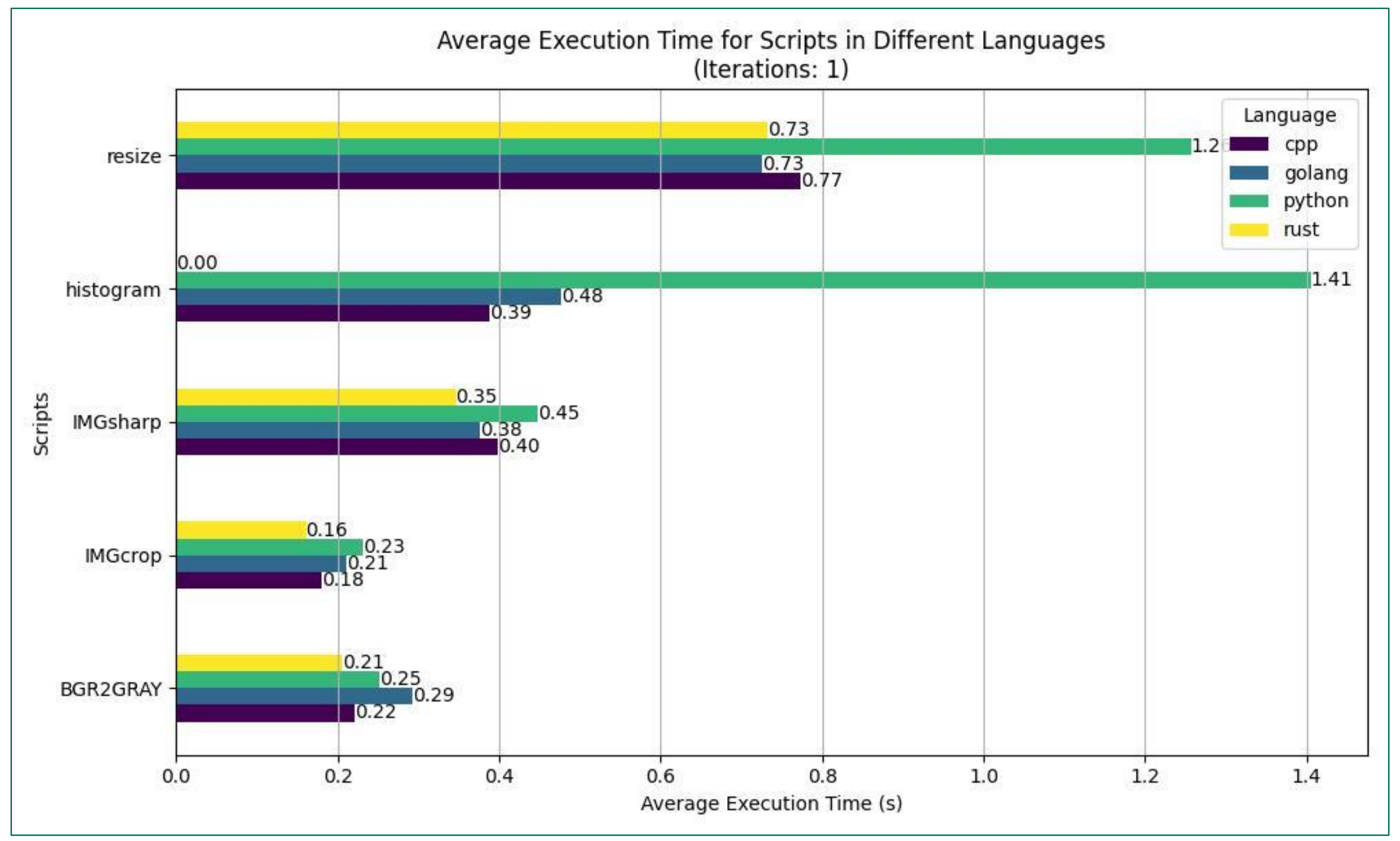


*Figure 3: Single Run Benchmark Results*

Across the single run benchmarks, Rust, C++, and Golang exhibit closely aligned execution times, notably in functions like resizing, showcasing similar efficiency levels and approaches in image processing tasks. Conversely, Python stands out with notably longer execution times in operations like histogram processing, suggesting potential performance concerns or implementation intricacies specific to Python's handling of these tasks. These disparities underscore the need for iterative runs to comprehensively assess the consistency and nuances in performance across these languages, enabling a deeper understanding of their individual strengths and weaknesses in image processing.

**Iterative Run**

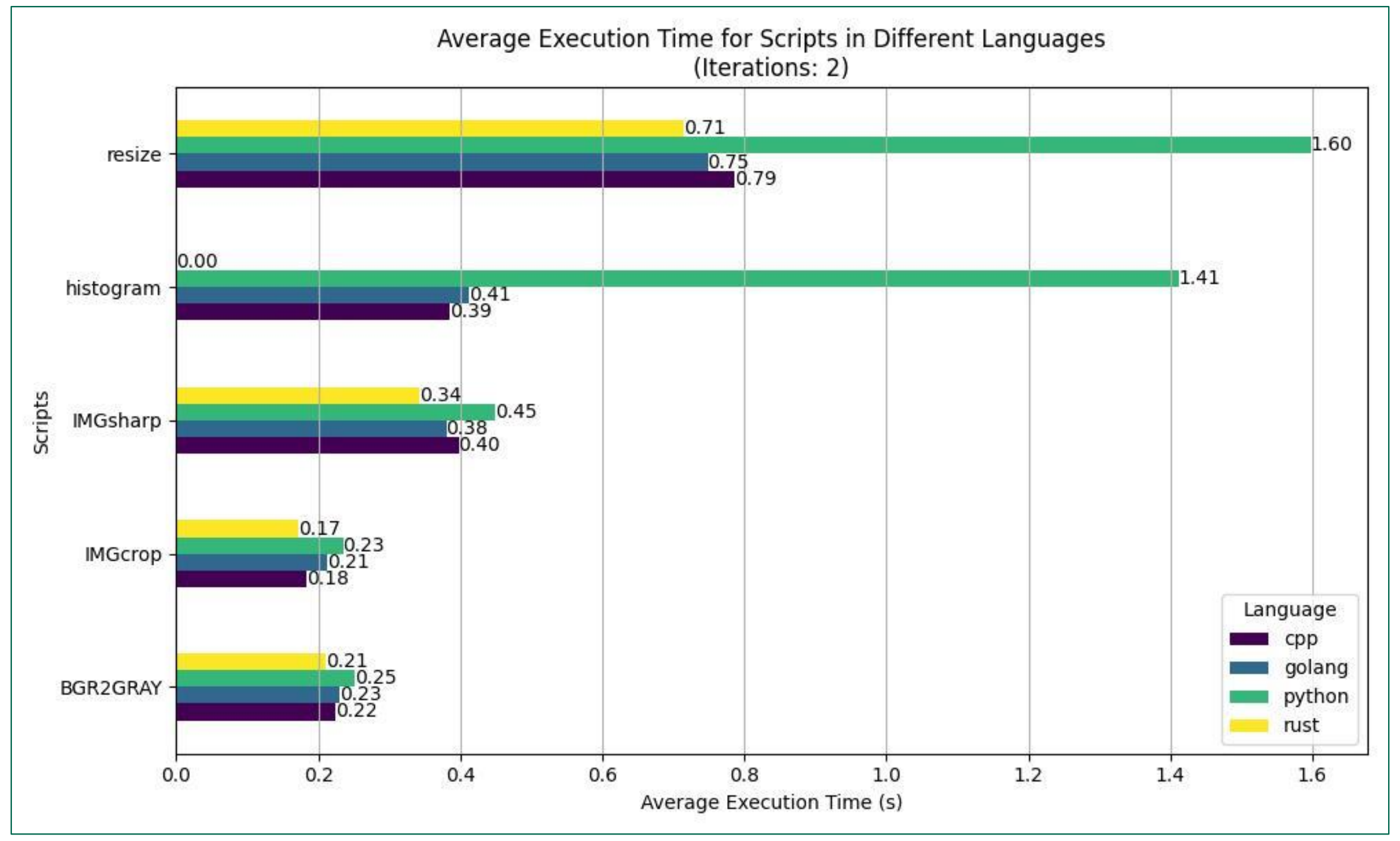


*Figure 4: Iterative Run Benchmark Results*

In the iterative run results, the Python scripts exhibit significant variability in execution times across iterations, notably seen in histogram processing and image resizing, highlighting inconsistent performance in these operations. Despite this, the Rust, C++, and Golang scripts demonstrate remarkable consistency and proximity in their execution times, showcasing similar efficiency across iterations. Notably, the Rust histogram operation was unsuccessful in both iterations, indicating a potential issue or limitation within the Rust script for this specific operation. The parallel efficiency among Rust, C++, and Golang reaffirms their closely aligned performance and reinforces the need for deeper analysis to understand any nuanced differences between these languages in image processing tasks.

## Conclusion

The benchmarking efforts revealed significant performance variations among Python, Rust, C++, and Golang in OpenCV image processing on the Raspberry Pi 4. Python demonstrated inconsistent execution times across iterations, notably in histogram and image resizing operations. Conversely, Rust, C++, and Golang exhibited consistent and competitive efficiency, closely aligned in their execution times. The iterative approach underscored the need for robust benchmarking to capture language-specific performance nuances accurately. Selecting the optimal language hinges on task-specific features. However, when comparing between C++, Rust, and Golang, Python's performance inconsistencies make it a challenging recommendation.

## References

[1]"OpenCV: Performance Measurement and Improvement Techniques," docs.opencv.org. https://docs.opencv.org/3.4/dc/d71/tutorial_py_optimization.html