CHAPTER V
CONCLUSION

5.1. Conclusion

Matrix multiplication testing by applying a two-dimensional array using green threads on Go on the order of 2000×2000 recorded the fastest execution time of 780.3 ms, demonstrating the language's superiority in CPU-Bound workload resolution. However, in the PageRank algorithm which emphasizes communication-intensive data synchronization, Rust appears more stable and efficient with an average acceleration ratio of 1.74× at epsilon 0.1. Rust's advantage continues in the Aho-Corasick algorithm-based REST API server scenario, where Rust recorded an average throughput of up to 1.76× higher at 199,271.65 RPS than Go and was able to maintain a deterministic tail latency (P_{99}) at 1.75 ms due to the absence of interruptions of the Garbage Collector cycle. Beyond runtime performance, the experiment also uncovered a contrasting dichotomy in development efficiency. The Rust compiler (rustc) is proven to be 54.3% to 78.6% faster in compile times than Go making it highly responsive to agile cycles, while Go's built-in binary structure structure consistently results in a binary file artifact size that is 15.3% to 79.1% more compact across test scenarios than Rust.

5.2. Suggestions

For further research development in the field of software engineering and distributed system performance evaluation, some suggestions that can be submitted are as follows:

1. **Expansion into the realm of Real Distributed Infrastructure**: Further research is suggested to move the test environment from an on-premises environment (bare-metal loopback testing) to a true distributed cloud native network architecture (such as Kubernetes clusters on AWS or Google Cloud Platform). It is important to test the robustness of the Goroutine Go and Async-Await Rust concurrency models in the face of internet network latency (jitter), I/O packet volatility, and non-deterministic inbound HTTP concurrency loads (inbound requests).

2. **High-Level Asynchronous Sync Library Utilization**: For developers focused on purely I/O-bound web applications (not numerical computing), it is recommended to explore the use of advanced external libraries in the Rust ecosystem such as Tokio Runtime and Actix-Web, and compare them to the performance improvements of Native Goroutine in the latest version of Go Compiler to get an industrial architecture optimization blueprint.
3. **Detailed RAM Depth Profiling Analysis**: It is recommended to add a memory-specific test instrument (such as *Valgrind*, *pprof*, or *Heaptrack*) to quantitatively record RAM consumption graph fluctuations (heap allocation spikes) when Go's Tricolor Mark-and-Sweep automation engine is actively working to match the stability of Rust's linear memory allocation line.