# Sustainable Image Processing for Digital News Platforms: Evaluating Go Concurrency Models for Efficient Media Workloads

*Hilmi Ra'if* Avicenna[1*], *Andi* Widiyanto[2], and *Rofi Abul* Hasani[1]

[1]Departement of Informatics Engineering, Universitas Muhammadiyah Magelang, Magelang Indonesia
[2]Departement of Information Technology, Universitas Muhammadiyah Magelang, Magelang, Indonesia

**Abstract.** The rapid expansion of visual content in digital news media has introduced significant computational challenges for image processing systems. This study evaluates the effectiveness of Go's concurrency mechanisms to address these bottlenecks, specifically through the implementation of a Worker Pool architecture. To validate its capability, the proposed model was benchmarked against a standard sequential process and a naive unbounded concurrency approach using the IMAGINE dataset under tiered load scenarios. The analysis focuses on execution time, memory stability, and CPU efficiency. The results demonstrate that Go's concurrency implementation using the Worker Pool pattern successfully maximizes system throughput, achieving a 3.74x speedup over the sequential baseline. Furthermore, unlike the naive approach which suffered from critical Out-of-Memory (OOM) failures, the Worker Pool maintained higher stability (CV 0.34%) and controlled resource usage. This study confirms that Go's concurrency, when implemented with a bounded strategy, is a highly effective solution for high-performance news portal systems.

## 1 Introduction

In the current digital media era, the popularity of photojournalism continues to rise alongside the demand for rapid information [1]. Images in modern news articles no longer serve merely as illustrations but act as elements that complement textual information, which is reflected in the increasing number of news articles incorporating multiple images as a key element of information delivery [2]. Digital news platforms such as the *Wall Street Journal* collected thousands of articles containing visual content, specifically 148,823 articles with photos over a 12-year period, demonstrating the massive scale of visual data in news media [1]. This scale is further emphasized by the NewsStories dataset, which successfully collected 31 million news articles with 22 million images from global

* Corresponding author: andi.widiyanto@ummgl.ac.id

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

https://doi.org/10.1051/e3sconf/202670603008

news platforms [3]. These platforms highlight the high volume of visual content that must be managed in modern digital news operations.

From a technical perspective, processing large numbers of images efficiently becomes difficult because traditional sequential image processing techniques cannot handle large-scale image datasets and computationally intensive algorithms [4]. A major limitation of this conventional sequential method is that it halts the active processing thread whenever data is being read or written, which prevents the processor from operating at its full potential and results in wasted computational power [5]. This issue is exacerbated in applications requiring intensive I/O operations with large data volumes, where I/O bottlenecks become a significant limiting factor. Consequently, minimizing the time users wait for a response is critical, necessitating the implementation of simultaneous data fetching and delivery mechanisms to accelerate the overall service speed [6].

Although modern concurrency models offer great potential for optimization, a review of existing studies highlights a fragmentation of solutions. On one hand, research in image processing, such as by Al-Neam et al. [4], focuses on optimizing CPU and GPU parallel computing for computationally intensive algorithms, but does not address I/O bottleneck handling as a primary contribution in an integrated concurrent processing design. Conversely, studies such as Hameed et al. [6] developed asynchronous concurrency-based solutions to address I/O-bound problems in distributed storage networks, but did not address the aspect of intensive computational processing in the image processing domain. On the other hand, the implementation of Go concurrency models analyzed by Gutmann and Rinner [7] in the context of multi-drone systems and Zhao et al. [8] for formal semantics development demonstrates the potential of goroutines and channels for concurrency, yet remains limited to those domains. Consequently, research specifically applying Go concurrency models to build image processing architectures that address both CPU-bound and I/O-bound bottlenecks in an integrated manner within the context of news portals remains limited.

The Go concurrency model offers a modern approach to these challenges by featuring goroutines as light-weight user-level threads and channels as a communication mechanism between goroutines [7,8]. Go employs the Communicating Sequential Processes (CSP) model, which facilitates data exchange and synchronization between threads using channels rather than shared memory locks [7]. Zhao et al. [8] emphasize that the efficiency of Go stems from its scheduler, which handles complex runtime interactions behind the scenes. This abstraction is vital for building robust applications that require intensive concurrent processing without burdening the developer with manual thread synchronization. The Go concurrency model also allows the Go runtime scheduler to distribute tasks to multiple CPU cores and has been proven to optimize hardware usage and significantly shorten processing delays [6].

Despite these advantages, improper use of the Go concurrency model can lead to goroutine leaks or partial deadlocks [8,9]. Leaks typically arise when a goroutine enters an indefinite wait state for a channel signal that never occurs. This state prevents the runtime from releasing the goroutine's stack and variables, leading to accumulated memory consumption that can eventually cause out-of-memory errors [9]. On the other hand, in concurrent Go application development, structured design patterns such as the worker pool represent an effective approach to limiting the number of goroutines

running simultaneously and distributing the workload evenly to a fixed number of workers [10]. To address these inefficiencies and bridge the gap between theoretical concurrency models and practical application, this study conducts a quantitative evaluation of image processing architectures within news portal systems. While this research focuses on news media, the proposed bounded concurrency architecture addresses universal I/O bottlenecks applicable to other data-intensive environments, such as distributed storage networks [6] and e-commerce platforms requiring stable resource management. The main contributions of this work are:

a. Empirical Performance Analysis: This study provides a direct comparison between Sequential, Naive Concurrent, and Worker Pool models.
b. Validation of Bounded Concurrency: It demonstrates that the Worker Pool model achieves a significant speedup (3.74x) over the baseline while preventing Out-of-Memory (OOM) crashes found in naive approaches.
c. Resource Behavior Characterization: The research identifies critical failure thresholds, proving that controlled worker allocation is essential for memory safety in containerized environments.

## 2 Method

This research applies a quantitative experimental method with a benchmarking approach to evaluate system performance in an objective, standardized, and reproducible manner. The primary objective is to measure the impact of implementing concurrency models on time efficiency and resource stability. Meanwhile, load testing methods are applied to measure system response under various load conditions to determine how the software behaves under varying workloads.

### 2.1 Test environment

To ensure internal validity and consistency of experimental results, all test scenarios are executed in an isolated container-based environment using Docker technology to ensure computational reproducibility [11]. Testing is performed on hardware equipped with an AMD Ryzen 7 processor and the Ubuntu 22.04 LTS operating system running within the Windows Subsystem for Linux (WSL). The detailed configuration of the hardware and runtime environment is summarized in **Table 1**.

**Table 1**. Experimental environment specifications

| Component | Specification |
|---|---|
| Processor (CPU) | AMD Ryzen 7 5800H (Sequential: 1 core pinned; Concurrent: 4 cores pinned) |
| Memory (RAM) | 16 GB DDR4 Host (Container limit: 4 GB Hard Limit) |
| Operating System | Windows 10 (via WSL 2 - Ubuntu 22.04 LTS) |
| Container Engine | Docker Engine 27.1.1 (Build 6312585) |
| Runtime & Base Image | Go 1.24.12 / Alpine 3.19.9 |
| Processing Engine | libvips 8.15.0 |
| Message Broker | RabbitMQ 3.13.7 |

Computational resources for the experimental units are strictly limited at the container level using Control Groups (cgroups) features, where memory allocation is set to 4 GB, identified as a standard configuration on various public cloud platforms. A key distinction is applied to CPU allocation. The sequential model is strictly limited to a single processor core (single-core) using CPU Pinning techniques, as speedup measurements must compare multi-core performance against a valid single-core baseline [12]. The application of CPU Pinning here is important to mitigate result distortion and minimize operating system overhead. Conversely, the concurrent models are allocated a full 4 cores to avoid operating system scheduling overhead that often distorts measurements on small instances.

### 2.2 Dataset and workload scenarios

This study utilizes a subset of the IMAGINE dataset [13], consisting of 100 selected image files with a total volume of 2.04 GB. These images were sourced from three professional camera devices: Nikon D5, Nikon D810, and Nikon Z7 II. To analyze system performance under various conditions, this study applies a stepwise approach where the processing of one image file is defined as a single unit of request. Workload variations begin with 1 image as a baseline measurement, increasing to a load of 10 images to observe system response when processing queues begin to form (measurable contention), and up to 50 and 100 images to test system resilience (sustained parallel demand) and identify scalability boundaries.

### 2.3 Processing model implementation

To ensure a comprehensive evaluation, the experimental prototype is designed with a distributed architecture consisting of a Message Broker (RabbitMQ) for workload distribution, an Image Service for execution, and File Storage for I/O operations. This design utilizes a producer-consumer pattern where tasks are queued before processing, ensuring that the measurement focuses strictly on the image processing performance without interference from request generation latency, as illustrated in **Fig. 1**.

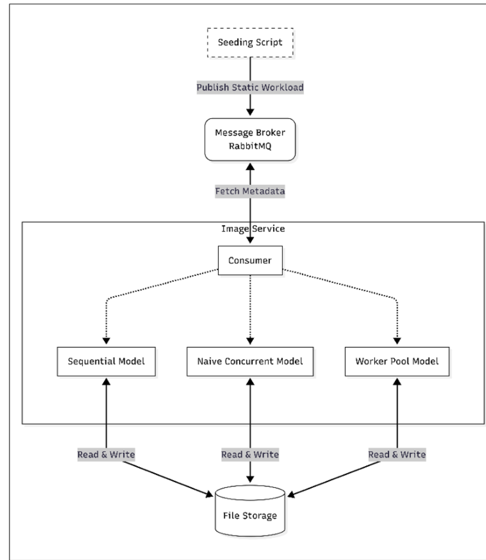


**Fig. 1**. Design of system architecture prototype

Three functional prototypes were implemented within this architecture to compare their performance characteristics. The first is the Sequential Processing Model, which executes tasks linearly, where a new task starts only after the previous task is completely finished. It serves as a valid control for calculating parallel processing efficiency [12]. The linear flow of this model, which blocks execution until each read-process-write cycle is complete, is depicted in **Fig. 2**.

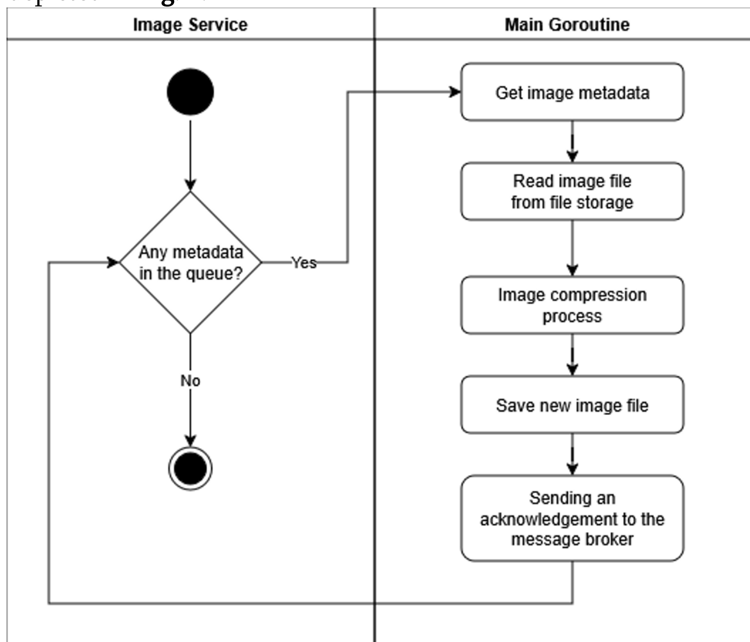


**Fig. 2**. Sequential model diagram

The second prototype is the Naive Concurrent Model. This model applies the principle of unbounded concurrency, where the system directly allocates a new execution unit

https://doi.org/10.1051/e3sconf/202670603008

(goroutine) for every incoming task without an intermediate queue. While this maximizes theoretical parallelism, it poses significant risks of resource exhaustion [9]. The logic for this approach, where independent goroutines are spawned for every image, is visualized in **Fig. 3**.

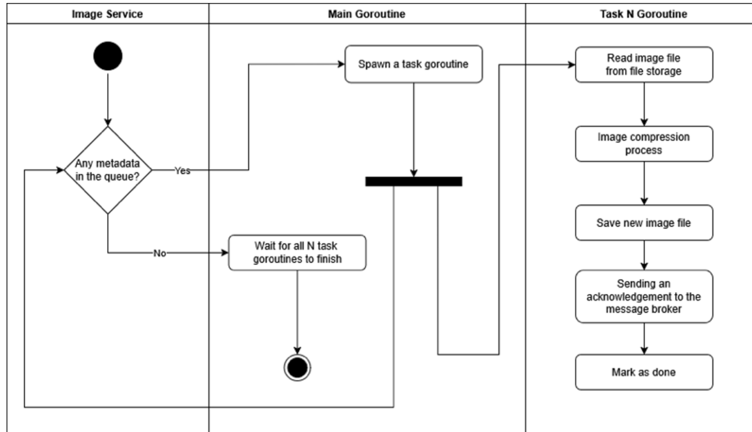


**Fig. 3**. Naive concurrent model diagram

Data processing utilizes the core technology of the WebP lossy compression format due to its ability to produce smaller file sizes compared to JPEG while maintaining comparable visual quality. The libvips library is used to support conversion to WebP format as it is known for high performance and efficient memory usage with a stream processing model [14].

Finally, the Worker Pool Concurrent Model serves as the proposed solution, utilizing a fixed number of execution units (workers) that operate persistently during the process lifecycle. Workload distribution is managed through a buffered channel to limit active concurrency [10]. This structured workflow, designed to balance CPU utilization and memory stability, is shown in **Fig. 4**.

 https://doi.org/10.1051/e3sconf/202670603008

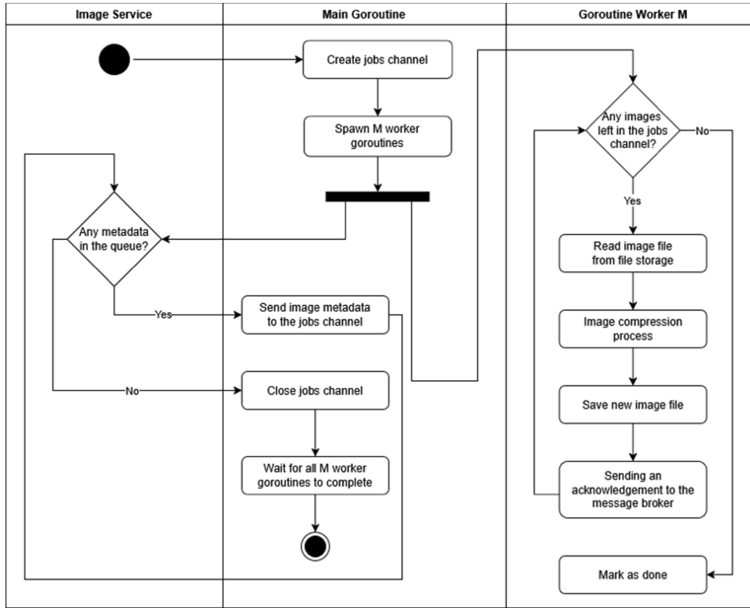


**Fig. 4**. Worker pool model diagram

## 2.4 Metrics and measurement procedures

Primary data in the form of total execution time, CPU utilization, and memory allocation are collected programmatically through system monitoring functions. The primary metric is Total Execution Time (T), measured in seconds (s) using the built-in Go time library. This value is used to calculate Speedup (Sp), defined as the ratio of sequential execution time (Ts) to parallel execution time (Tp) [12]. The formula used is:

$$S_p = \frac{T_s}{T_p} \quad (1)$$

Secondary metrics include CPU utilization, measured in percentage (%). The underlying data is obtained from the external gopsutil library, a third-party Go library providing an interface to retrieve system resource statistics such as CPU, memory, disk, and network regularly. CPU Utilization (U) is defined as the ratio of busy time (B) to the total observation time period (T) [15]. This metric validates the efficiency of resource usage across multiple cores:

$$U = \frac{B}{T} \ (2)$$

Additionally, Peak Memory Allocation (Mpeak) is measured in Megabytes (MB) through a periodic monitoring approach to characterize memory performance profiles under varying workload conditions. It is calculated as the maximum value derived from memory usage samples (mt) taken at time t:

$$M_{peak} = max(m_t) \quad (3)$$

To ensure the validity and reliability of the data, statistical analysis is performed to measure consistency. Standard Deviation (σ) is calculated to quantify the amount of

variation or dispersion of the execution time data points (xi) from their mean (μ) for the number of samples (N):

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i-\mu)^2}{N-1}} \quad (4)$$

Finally, to evaluate performance stability relative to the mean, the Coefficient of Variation (CV) is calculated. A lower CV percentage indicates more stable and predictable system performance [15]:

$$CV = \frac{\sigma}{\mu} \times 100\% \quad (5)$$

Each benchmark scenario is executed five times to mitigate the impact of execution environment performance variability and ensure statistically significant results.

## 3 Results and discussion

This section presents the experimental findings regarding the performance of the three image processing models-Sequential, Naive Concurrent, and Worker Pool-under varying workload scenarios. The analysis focuses on execution time, resource efficiency, and system stability, followed by a discussion on their implications in the context of news portal systems.

### 3.1 Results

The total execution time (T) for processing the IMAGINE dataset varied significantly across the three models. As detailed in **Table 2**, the baseline Sequential model exhibited a linear increase in execution time, reaching 387.99 seconds at the maximum workload of 100 images.

**Table 2**. Experimental results comparing model performance and resource usage

| Workload | Processing Model | Execution Time (s) (Mean ± SD) | Speedup (x) | CPU Util. (%) (Mean) | Peak Memory (MB) (Max) | Experiment Outcome |
|---|---|---|---|---|---|---|
| 1 Image | Sequential | 1.73 ± 0.03 | 1.00 | 100.40 | 156.63 | Success |
| | Naive | 1.66 ± 0.02 | 01.04 | 105.38 | 156.52 | Success |
| | Worker Pool | 1.68 ± 0.05 | 01.03 | 105.40 | 156.53 | Success |
| 10 Images | Sequential | 16.59 ± 0.06 | 1.00 | 100.00 | 156.92 | Success |
| | Naive | 4.99 ± 0.06 | 3.32 | 389.83 | 1,226.29 | Success |
| | Worker Pool | 5.07 ± 0.03 | 3.27 | 343.41 | 533.08 | Success |
| 50 Images | Sequential | 138.33 ± 0.26 | 1.00 | 99.99 | 334.40 | Success |
| | Naive | - | - | - | - | Crash (OOM) |

| | Worker Pool | 40.10 ± 0.09 | 3.45 | 361.87 | 1,014.92 | Success |
|---|---|---|---|---|---|---|
| 100 Images | Sequential | 387.99 ± 1.55 | 1.00 | 99.99 | 397.31 | Success |
| | Naive | - | - | - | - | Crash (OOM) |
| | Worker Pool | 103.73 ± 0.36 | 3.74 | 394.16 | 1,228.78 | Success |

In contrast, the concurrent models demonstrated a substantial reduction in processing time. The Naive Concurrent model achieved the fastest time of 4.99 seconds at 10 images but failed to complete higher workloads. The Worker Pool model consistently completed all tasks, recording 103.73 seconds at the maximum workload. The distinct performance gap between the sequential and concurrent approaches is visualized in **Fig. 5**.

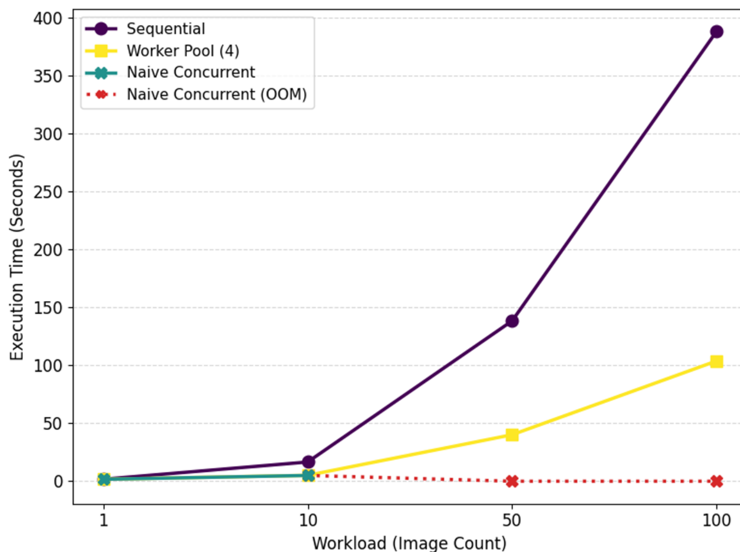


**Fig. 5**. The execution time comparison graph

Based on the execution time data in **Table 2**, the speedup ratio was calculated. The Worker Pool model achieved a maximum speedup of 3.74x at the highest workload (100 images) compared to the sequential baseline. This value approaches the theoretical limit for the 4-core processor used. While the Naive model showed a speedup of roughly 3.32x at a load of 10 images, it could not sustain this performance at higher loads due to system failure.

Regarding resource utilization, CPU measurements illustrated in **Fig. 6** validate the experimental environment constraints. The Sequential model maintained a utilization average of ~100% (single-core), whereas the Worker Pool model effectively utilized the multi-core environment with an average of 394.16% at maximum load. However, a critical distinction is shown in memory performance **Fig. 6**.

 https://doi.org/10.1051/e3sconf/202670603008

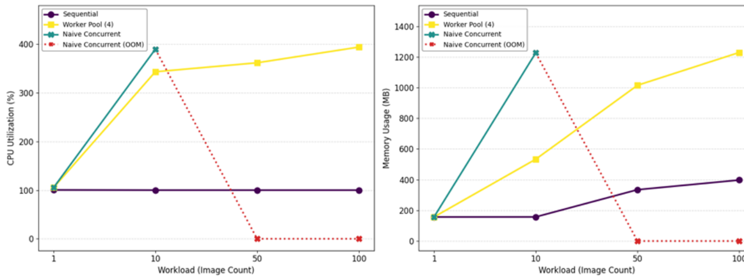


**Fig. 6**. The CPU usage and memory (RAM) allocation comparison chart

The Naive Concurrent model exhibited a linear surge in memory usage, reaching a peak of 1,226.29 MB with just 10 images before crashing at 50 images due to Out-of-Memory (OOM) errors. The Worker Pool model successfully capped memory usage (Max: 1,228.78 MB at 100 images), demonstrating a controlled resource profile even under heavy load. Furthermore, stability analysis using the Coefficient of Variation (CV) revealed that the Worker Pool model offered consistent performance with a very low CV of 0.34% at maximum load, whereas the Naive model showed significantly higher variability (1.18%) at lower loads before failing.

### 3.2 Discussion

The results confirm that concurrent models significantly outperform the sequential approach in image processing tasks involving high-resolution datasets like IMAGINE. This finding aligns with the premise established by Al-hayanni et al. [12], where parallel execution on multi-core systems overcomes the limitations of serial processing. The Sequential model suffered from the "blocking" phenomenon described by Lee and Park [5], where the CPU remained idle during I/O operations (reading/writing files). In contrast, the Worker Pool model effectively interleaved I/O and CPU-bound tasks, validating the efficiency of the Go scheduler in distributing tasks as noted by Hameed et al. [6].

A critical finding of this study is the trade-off between raw speed and resource stability, as clearly depicted in **Fig. 6**. While the Naive Concurrent model provided fast execution at low loads, its memory consumption grew linearly with the workload, leading to crashes at 50 and 100 images. This behavior supports the warning by Saioc et al. [9] regarding "goroutine leaks" and resource exhaustion in unbounded concurrency. The Naive model forces the runtime to schedule excessively large numbers of threads simultaneously, leading to fatal memory pressure.

Conversely, the proposed Worker Pool model demonstrated higher scalability. By limiting the number of active workers (N=4), this model maintained a stable memory footprint relative to the throughput. This result clarifies the theoretical advantage of structured concurrency patterns discussed by Serdar [10]. The Worker Pool proves to be the optimal solution for news media platforms, as it offers a speedup of 3.74x with the stability required for continuous operation, effectively preventing the out-of-memory risks associated with processing massive datasets like NewsStories [3]. Moreover, the stability analysis (CV) indicates that the Worker Pool model is highly predictable, which is a key metric for microservices under variable workloads. The findings suggest that the

overhead of context switching in the Naive model introduces latency jitter and instability, whereas the Worker Pool's bounded concurrency ensures consistent throughput.

## 4 Conclusion

This study evaluated the performance of Go concurrency models for processing high-resolution images within digital news media systems. The experimental results demonstrate that implementing the Worker Pool model significantly enhances performance, achieving a maximum speedup of 3.74x compared to the sequential baseline, thereby validating the efficiency of Go's concurrency on multi-core hardware. Crucially, the Worker Pool architecture proved to be the most robust solution; unlike the Naive Concurrent model, which suffered from critical Out-of-Memory (OOM) failures under high workloads (50 and 100 images), the Worker Pool successfully managed memory consumption within the allocated 4GB limit. Furthermore, this model exhibited higher stability with a negligible Coefficient of Variation (0.34%), ensuring predictable resource usage which is essential for production environments.

Consequently, this research establishes that Worker Pool is the optimal strategy for high-throughput image processing, effectively resolving simultaneous CPU and I/O bottlenecks without risking resource exhaustion. Future research should extend this evaluation to distributed cluster environments using orchestration tools like Kubernetes to assess horizontal scalability across multiple nodes, as well as exploring the impact of varying compression algorithms on system throughput.

Despite these findings, this study has several limitations. First, the evaluation was conducted using the IMAGINE dataset, which, while providing high-resolution images, represents a homogeneous workload. In a real-world news portal scenario, traffic is often heterogeneous, consisting of mixed request types that may introduce different resource contention patterns. Second, the experiment was limited to a single-node environment using Docker containers. While this effectively isolates performance variables, it does not fully capture the network latency and distributed coordination overhead present in multi-node clusters. Future research should address these gaps by testing mixed workloads and scaling the Worker Pool architecture across distributed infrastructure.

## References

1. Obaid, K.; Pukthuanthong, K. A Picture Is Worth a Thousand Words: Measuring Investor Sentiment by Combining Machine Learning and Photos from News. *J. financ. econ.* **2022**, *144*, 273–297, doi:10.1016/j.jfineco.2021.06.002.
2. Wang, J.; Zheng, J.; Yao, S.; Wang, R.; Du, H. TLFND: A Multimodal Fusion Model Based on Three-Level Feature Matching Distance for Fake News Detection. *Entropy* **2023**, *25*, 1533, doi:10.3390/e25111533.
3. Tang, P.; Zhu, S.; Alatas, B. Improving News Recommendation Accuracy Through Multimodal Variational Autoencoder and Adversarial Training. *IEEE Access* **2025**, *13*, 85269–85278, doi:10.1109/ACCESS.2025.3568514.

4. Mohammed W. Al-Neam A Novel Parallel Approach to Image Processing for High-Performance Computing. *J. Inf. Syst. Eng. Manag.* **2025**, *10*, 721–729, doi:10.52783/jisem.v10i7s.985.
5. Lee, S.; Park, J. Comparative Performance Analysis of I/O Interfaces on Different NVMe SSDs in a High CPU Contention Scenario. *IEICE Trans. Inf. Syst.* **2024**, *E107.D*, 2024EDL8012, doi:10.1587/transinf.2024EDL8012.
6. Hameed, Z.; Pahl, C.; Berenjestanaki, M.H.; Van Thillo, I. A Secure and Reliable Distributed Storage System with Multi-Factor Trust-Aware Workflow Scheduling. *Cluster Comput.* **2025**, *28*, 1031, doi:10.1007/s10586-025-05625-1.
7. Gutmann, M.; Rinner, B. Multidrone Mission Execution With EAMOS: From Text to Mission. *IEEE Access* **2023**, *11*, 125460–125491, doi:10.1109/ACCESS.2023.3330652.
8. Zhao, C.; Liu, Q.; Hu, Z.; Yu, Z.; Wang, D.; Meng, B. K-Go: An Executable Formal Semantics of Go Language in K Framework. *IET Blockchain* **2023**, *3*, 61–73, doi:10.1049/blc2.12024.
9. Saioc, G.-V.; Lange, J.; Møller, A. Automated Verification of Parametric Channel-Based Process Communication. *Proc. ACM Program. Lang.* **2024**, *8*, 2070–2096, doi:10.1145/3689784.
10. Serdar, B. *Effective Concurrency in Go Develop, Analyze, and Troubleshoot High Performance Concurrent Applications with Ease*; Kinnari Chohan, J.P., Ed.; 1st ed.; Birmingham, 2023; ISBN 978-1-80461-907-0.
11. Moreau, D.; Wiebels, K.; Boettiger, C. Containers for Computational Reproducibility. *Nat. Rev. Methods Prim.* **2023**, *3*, 50, doi:10.1038/s43586-023-00236-9.
12. Al-hayanni, M.A.N.; Rafiev, A.; Xia, F.; Shafik, R.; Romanovsky, A.; Yakovlev, A. PARMA: Parallelization-Aware Run-Time Management for Energy-Efficient Many-Core Systems. *IEEE Trans. Comput.* **2020**, *69*, 1507–1518, doi:10.1109/TC.2020.2975787.
13. Bernacki, J.; Scherer, R. Algorithms and Methods for Individual Source Camera Identification: A Survey. *Sensors* **2025**, *25*, 3027, doi:10.3390/s25103027.
14. Cupitt, J.; Martinez, K.; Fuller, L.; Wolthuizen, K. The Libvips Image Processing Library. *Electron. Imaging* **2025**, *37*, 178-1-178–7, doi:10.2352/EI.2025.37.12.HPCI-178.
15. Gregg, B. *Systems Performance Enterprise and the Cloud*; Second Edi.; Addison-wesley professional computing series, **2021**; ISBN 978-0-13-682015-4 0-13-682015-8.