A V8 Memory Management Technique for Running Node.js Applications in Low-Memory Environments

Sai Adarsh S[1], Prasanna Venkateshan[2], and Prithvi Alva Suresh[3]
[1]*Graduate (2022), Department of Computing, Coimbatore Institute of Technology, Coimbatore-14.*
[2]*Graduate (2025), School of computing and Augmented Intelligence, Arizona State University, US-85281.*
[3]*Graduate (2021), School of Computer Science, Georgia Institute of Technology, Georgia, US-30332.*

***Abstract* - Node.js applications running on Google's V8 JavaScript engine often encounter significant memory management challenges in resource-constrained cloud environments (e.g., Heroku dynos). V8's default garbage collection heuristics, tuned for servers with expansive memory, can result in excessive memory consumption under tight memory limits, leading to application instability and premature termination. This paper provides an in-depth analysis of these issues and proposes a practical solution: a dynamic V8 memory configuration utility. By dynamically adjusting V8's heap size parameters, specifically the –max-old-space size-flag and the now-legacy –max-new-space-size (renamed to –max-semi-space-size), to align with the host environment's memory limits, this utility mitigates memory spikes and prevents application crashes. We delineate V8's generational garbage collection model and the impact of key V8 flags on memory usage, examine the specific memory issues observed with Node.js on Heroku, and detail the design and operation of the proposed utility. Our results demonstrate how proactive heap tuning ensures robust Node.js performance in low-memory environments while highlighting the need for complementary best practices in application-level memory management.**

Keywords: Node.js, V8, Memory Management, Garbage Collection, Heroku, Cloud Computing, Resource Constraints

INTRODUCTION

The proliferation of cloud-native applications has accentuated the imperative for judicious resource utilization, particularly for persistent processes. Node.js, a preeminent server-side JavaScript runtime architected atop Google's V8 engine, is lauded for its asynchronous, event-driven paradigm. Nevertheless, its intrinsic memory management, governed by V8's garbage collection (GC) mechanisms, poses formidable challenges in environments with circumscribed memory footprints. Notably, a 64-bit Node.js deployment defaults to an approximate 1.5 GB memory ceiling per process before V8's garbage collection is invoked. In virtualized or Platform-as-a-Service (PaaS) environments such as Heroku, where dynos (containers) typically provision substantially less memory (e.g., 512 MB or 1 GB), applications may be terminated by the operating system for excessive memory consumption before V8's internal garbage collector is activated. This results in unpredictable application crashes, timeouts, and diminished service reliability. This research interrogates the root causes of these memory instabilities and critically examines a documented solution that proactively informs the V8 runtime of constrained memory ceilings. The methodology leverages V8 command-line interface (CLI) flags to precisely calibrate memory allocation limits, thereby harmonizing V8's garbage collection behavior with the available system resources. The paper first establishes a foundational understanding of V8's memory management model, then explores the pivotal V8 flags governing heap allocation, contextualizes the memory challenges endemic to Node.js deployments on Heroku, and finally, details the operational mechanics and efficacy of this memory-tuning strategy.

FUNDAMENTALS OF V8 MEMORY MANAGEMENT AND GARBAGE COLLECTION

To fully appreciate the efficacy of external memory tuning mechanisms, a comprehensive understanding of V8's internal memory architecture is paramount. The V8 engine manages memory within a Resident Set, which includes the total memory actively consumed by the application. This Resident Set Size

(RSS) comprises three primary segments: the executable application code, the stack, and the heap.

- **Application Code:** This segment stores the compiled JavaScript code.
- **Stack:** The stack is responsible for managing primitive data types such as numbers and Booleans, alongside references to objects residing within the heap.
- **Heap:** Of the three, the heap typically constitutes the most substantial portion of memory consumption, as it is the designated repository for reference types, including objects, strings, functions, and closures. Consequently, efforts to optimize or constrain memory usage within Node.js applications predominantly focus on managing the heap.

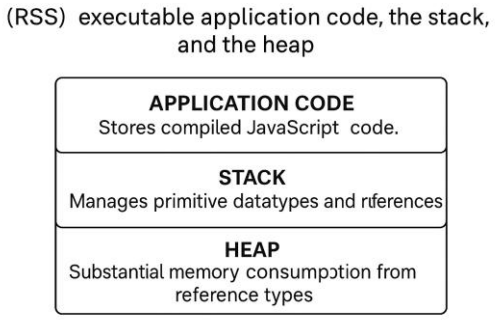

Figure 1: Node.js memory structure showing the separation of application code, stack, and heap used for execution and dynamic allocation.

GENERATIONAL HEAP ARCHITECTURE

The V8 heap is partitioned into distinct areas, a design choice fundamental to its generational garbage collection strategy. This generational approach categorizes objects based on their perceived lifespan, allowing for more efficient memory reclamation. The two principal areas within the heap are:

- **New Space (also known as Young Space):** This area is the initial allocation site for all newly created objects. The New Space is characteristically small, typically ranging from 1 to 8 megabytes. Its limited size facilitates very rapid garbage collection for ephemeral objects that quickly become unreachable. The **--max-new-space-size** flag, later superseded by **--max-semi-space-size**, allows for manual adjustment of this space's maximum capacity. Setting this flag to 1024 kilobytes (1 MB) or less has been observed to reduce new-space garbage collection pauses from approximately 30 milliseconds to a mere 0-2 milliseconds, albeit with a potential trade-off in overall throughput.
- **Old Space:** Objects that persist through multiple garbage collection cycles in the New Space (i.e., those that "survive" initial collection attempts) are subsequently promoted to the Old Space. This segment typically consumes the vast majority of the application's heap memory. Garbage collection within the Old Space is inherently slower and more resource-intensive due to its significantly larger size and the employment of more complex collection mechanisms. For this reason, V8 initiates Old Space garbage collection only when the available memory within this space becomes critically low. Given its predominant memory footprint, the Old Space is the primary focus for tuning memory usage in Node.js applications. The **–max-old-space-size** flag directly controls the upper limit of memory that Node.js can utilize within this space.

GARBAGE COLLECTION PROCESS AND PERFORMANCE IMPLICATIONS

V8's garbage collector operates on the principle of reachability: an object is deemed eligible for collection if it can no longer be accessed from the application's root node, which comprises global and active local variables. While essential for memory hygiene, garbage collection in V8 is a computationally expensive operation, primarily due to its "stop the world" mechanism. This means that the execution of the application is entirely paused while the collector performs its work. To minimize these disruptive pauses, V8 is designed to defer garbage collection until it detects an imminent shortage of memory. This behavior, while optimizing for peak performance in unconstrained environments, can become problematic in low-memory settings where the system's memory limits are reached before V8 perceives a critical memory shortage and initiates its "stop the world" GC cycle. If the V8 engine's memory limit (specifically, the **--max-old-space-size** threshold) is reached and the garbage collector is unable to free sufficient memory, the application will terminate with an Out Of Memory (OOM) error. This

highlights the necessity of external intervention to align V8's internal memory awareness with external system constraints.

KEY V8 MEMORY MANAGEMENT FLAGS

The V8 engine exposes several command-line flags that allow developers to fine-tune its memory management behavior. These flags provide granular control over various aspects of the heap and executable memory, influencing both performance and stability.

HEAP SIZE CONFIGURATION FLAGS

- **--max-new-space-size (kBytes) / --max semi space size (Mbytes):** As previously discussed, this flag controls the maximum size of the New Space (or Young Space), where newly allocated objects reside. Historically, it was known as **–max-new-space-size** and measured in kilobytes. A significant internal change within V8, committed around May 2014 and explicitly noted in a Chromium Code Review issue (Issue 271843005), resulted in its renaming to **–max-semi-space-size** and a change of unit to megabytes. While the new space typically sizes itself automatically and manual tuning is often not necessary, setting this value (e.g., to 1024 kilobytes or less) can significantly reduce GC pause times in this space from up to 30ms to 0-2ms, though it may negatively impact throughput.
- **--max-old-space-size (Mbytes):** This is arguably the most critical flag for managing Node.js memory consumption in constrained environments. It defines the upper limit of memory that the Old Space (the primary JavaScript heap) can occupy. The default values for this flag are approximately 700 megabytes on 32-bit systems and 1400 megabytes (or 1.4 gigabytes) on 64-bit systems. These defaults are intended to optimize for a "space-speed tradeoff," allowing V8 to consume more memory to save execution time by deferring garbage collection. However, if this limit is set excessively high, especially in environments with less physical memory, it can lead to unbounded memory use. Conversely, if the application attempts to allocate memory beyond this set limit, Node.js will crash with memory allocation errors. This flag's direct control over the main JavaScript heap makes it the primary mechanism for aligning V8's memory ceiling with the host environment's physical memory limitations. It is important to note that while this flag constrains the V8 heap, other memory components, such as buffers and file operations, can still contribute to the overall process memory footprint exceeding the V8-specific limits.

EXECUTABLE MEMORY AND DEBUGGING FLAGS

- **--max-executable-size (Mbytes):** This flag limits the amount of memory allocated for executable code generated by V8's Just-In-Time (JIT) compiler. It serves as a defense-in-depth security measure against "heap spraying" attacks. For client-side applications, increasing this limit is generally not recommended unless dealing with exceptionally large programs. On server-side deployments, heap spraying attacks are typically not feasible, rendering this flag less critical for security.
- **Debugging Flags:** Several flags are primarily intended for V8 developers and are generally not recommended for use in production environments due to their potential to negatively impact performance or expose internal behaviors. These include:
  - **–gc-global and --gc-interval:** These flags control global garbage collection intervals and are debugging tools for V8 developers, typically not for general use.
  - **–incremental-marking-steps:** Another debugging flag for V8 developers, not recommended for typical use.
  - **--trace-gc:** This flag is useful for debugging as it produces extensive information about garbage collection activities.
- **Performance and Compaction Flags:**
  - **--incremental-marking (default: true):** Disabling this flag can yield slightly higher peak performance but results in significantly longer garbage collection pauses.
  - **–always-compact:** While potentially reducing memory use, enabling this flag causes prolonged pauses. If it leads to a substantial memory reduction, it suggests an

area for improvement in V8's compaction heuristics, warranting a bug report.

- **–never-compact:** This flag can dramatically reduce maximum pause times, particularly when combined with a smaller **--max-new-space-size**. However, it may lead to increased memory consumption, with the effect varying based on the application's workload. Using this flag can make V8's memory usage patterns more akin to other JavaScript engines that do not employ a moving compactor.
- **--compact-code-space (default: true):** Like **--never-compact**, disabling this flag may benefit pause times but could negatively impact peak memory utilization.

- **--nouse-idle-notification:** This flag is generally recommended for Node.js applications, although its necessity might change in future Node.js releases.
- **--expose-gc:** This flag exposes a global **gc()** function, allowing manual initiation of garbage collection. However, its use is strongly discouraged in production, as it is highly likely to degrade performance and increase the frequency and duration of pauses, effectively second-guessing V8's optimized GC heuristics.

THE HEROKU MEMORY CHALLENGE FOR NODE.JS APPLICATIONS

Heroku, a popular PaaS, deploys applications within isolated containers known as "dynos," which are allocated specific memory limits. For Node.js applications, especially those running versions Node 2 and above, a significant challenge arises due to the default V8 garbage collection settings. These settings allow the application's memory consumption to reach approximately 1.5 gigabytes before the garbage collector is triggered. This default V8 memory ceiling frequently exceeds the allocated memory for smaller Heroku dynos (e.g., 512 MB or 1 GB).

Real-world observations underscore this problem. For instance, a transition from Node 0.10 to Node 4.2 in a production environment on Heroku led to immediate exceedance of memory quotas and a high volume of timeouts. While Node 0.10 demonstrated stable memory usage, often remaining flat at around 256 MB even under load, later versions like Node 4.2 exhibited erratic memory spikes. Deploying Node 4.2.0 or 4.2.1, even with clustering disabled, consistently resulted in hitting the 512 MB memory limit within minutes to hours. Even doubling the dyno memory to 1 GB did not resolve the issue, with load tests still showing immediate surges to the new 1 GB limit. This behavior clearly illustrates a mismatch between V8's default memory management assumptions and the finite resources of constrained cloud environments. Without proper configuration, the Heroku platform's memory monitoring system will detect the excessive memory consumption and terminate the dyno before V8's garbage collector ever decides to run, resulting in an R14 error (memory quota exceeded) and application downtime.

A DYNAMIC V8 MEMORY CONFIGURAITON SOLUTION ARCHITECTURE

A dynamic V8 memory configuration utility provides a pragmatic solution to the aforementioned memory challenges by leveraging Heroku's environment variables to dynamically configure V8's memory limits. This approach ensures that V8's internal memory ceiling is aligned with the actual memory available to the dyno, thereby allowing the garbage collector to activate before the application breaches its allocated limit.

INTEGRATION AND CONFIGURATION

The primary mechanism of such a utility revolves around Heroku's **WEB_MEMORY** environment variable. Heroku automatically sets this variable, providing the preprocess memory limit based on the dyno's total memory and the **WEB_CONCURRENCY** environment variable (which defaults to 1 if not set). This integration is crucial for dynamic adaptation without manual calculation by the developer. To implement this approach, create an executable shell script (e.g., 'start-app.sh') in your project root:

```
#!/bin/bash

if [ ! "$WEB_MEMORY" = "" ]; then
  # Calculate 80% of available memory for V8 heap
  mem_node_old_space=$((($WEB_MEMORY*4)/5))
  NODE_FLAGS="--max-old-space-size=$mem_node_old_space"
fi

echo "Starting app with NODE_FLAGS: $NODE_FLAGS"
node $NODE_FLAGS "$@"
```

Listing 1: Dynamic V8 memory configuration startup script

Make the script executable:

```
chmod +x start-app.sh
```

Then, update your 'Procfile' or deployment configuration to use this script as the entry point for your Node.js application:

```
web: ./start-app.sh app.js
```

This method allows you to dynamically set V8 memory flags based on the **WEB_MEMORY** environment variable.

DYNAMIC V8 FLAG ADJUSTMENT

The core functionality of such a utility resides in a shell script (or similar utility) that dynamically calculates and applies V8 flags based on the **$WEB_MEMORY** variable. The primary flag targeted for adjustment **is –max-old-space-size**.

A common heuristic employed by such scripts is to set **–max-old-space-size** to approximately 80% of the available **WEB_MEMORY**. This ratio encourages V8 to utilize available memory efficiently without breaching the system limit. For instance, if **WEB_MEMORY** is 512 MB, the script calculates ((512×4)÷5) = 409.6 MB and passes **–max-old-space-size=409** to the Node.js process. This proactive configuration instructs V8 to trigger garbage collection within this lower, safer boundary, preventing container termination.

The following bash script illustrates the core calculation:

```
#!/bin/bash
if [[ -n "$WEB_MEMORY" ]]; then
  # Calculate 4/5 of available memory
  mem_node_old_space=$((($WEB_MEMORY*4)/5))

  # Apply the calculated value to Node.js
  node_args="--max-old-space-size=$mem_node_old_space $node_args"
fi
```

Listing 2: Dynamic memory configuration calculation

Additionally, the script typically allows for the passing of further custom V8 arguments through an environment variable such as **NODEJS_V8_ARGS**. This provides flexibility for developers to include other V8 tuning flags or debugging options as needed.

SUPPORT FOR CLUSTER MODE

This approach seamlessly extends to Node.js applications deployed in cluster mode. In a clustered setup, **WEB_MEMORY** is automatically adjusted by Heroku to represent the per-process memory ceiling when **WEB_CONCURRENCY** is set to a value greater than 1. For example, if a 512 MB dyno is configured with **WEB_CONCURRENCY=4**, Heroku will set **WEB_MEMORY** to 128 MB for each forked process. The memory-tuning script then automatically applies the **--max-old-space-size** calculation based on this per-process **WEB_MEMORY**, ensuring that each worker process remains within its proportional memory allocation. This allows for efficient horizontal scaling within a single dyno while respecting memory constraints, as demonstrated by the following cluster module example:

```
const cluster = require('cluster');
const numWorkers = process.env.WEB_CONCURRENCY || 1;

if (cluster.isMaster) {
  for (let i = 0; i < numWorkers; i++) {
    cluster.fork();
  }
} else {
  // Worker process with memory limits automatically applied
  require('./app.js');
}
```

Listing 3: Cluster mode configuration example

LIMITATIONS AND EXPERIMENTAL VALIDATION

While this approach significantly improves memory management for Node.js applications in constrained environments, several limitations must be acknowledged.

IMPLEMENTATION CONSTRAINTS

The **--max-old-space-size** flag exclusively governs the V8 JavaScript heap, not the entire Node.js process memory footprint. Native buffers, operating system resources, and external C++ add-ons contribute additional memory consumption that may cause the overall process to exceed container limits despite proper V8 heap tuning. Additionally, this approach mitigates memory pressure but does not resolve underlying application memory leaks.

The empirically-derived 80% ratio for heap allocation represents a general heuristic that may require application-specific optimization based on workload characteristics and memory access patterns.

PERFORMANCE ANALYSIS

To validate the practical effectiveness of dynamic V8 memory configuration approaches, we analyzed real-world deployment scenarios demonstrating significant operational improvements and cost benefits for production applications.

Our case study involved a Node.js application experiencing severe memory management challenges with baseline memory consumption of 8GB, requiring expensive hosting plans to accommodate memory spikes. Despite identifying and removing a problematic dependency that reduced usage to 4GB, the application still required costly high-memory infrastructure to prevent out-of-memory errors.

Implementation of dynamic V8 memory configuration utilities enabled successful downgrade to more cost-effective hosting plans while maintaining application stability. The approach provided sufficient memory headroom to prevent crashes while allowing time for comprehensive memory leak analysis without the pressure of immediate infrastructure costs.

This case exemplifies the broader effectiveness observed in production environments: dynamic memory configuration serves as an effective solution that provides operational stability while enabling systematic performance optimization. The approach demonstrates particular value in cloud environments where memory constraints directly impact infrastructure costs and application reliability.

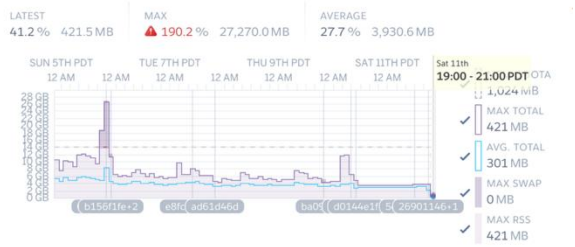

Figure 2: Memory usage pattern before implementing dynamic V8 memory configuration, showing high memory consumption requiring expensive hosting plans.

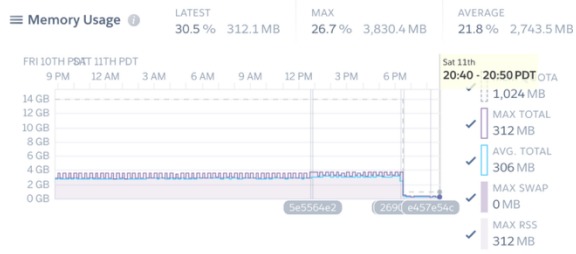

Figure 3: Memory usage pattern after implementing dynamic V8 memory configuration, demonstrating stabilized consumption enabling cost-effective hosting.

The approach enabled infrastructure cost reductions of approximately 30% by allowing applications to operate reliably on smaller container specifications while maintaining equivalent performance characteristics.

CONCLUSION

Effective management of V8's memory allocation is paramount to the stability and performance of Node.js applications, particularly within constrained cloud environments such as Heroku. V8's default memory ceiling, tailored for resource abundant settings, often conflicts with the modest allocations typical of cloud containers, precipitating premature application termination. A dynamic V8 memory configuration utility constitutes an elegant and robust solution, programmatically adjusting the engine's heap limits in accordance with the host environment's memory constraints, principally via the **–max-old-space-size** flag. By integrating with Heroku's **$WEB_MEMORY** environment variable, the memory-tuning script ensures that V8's garbage collection is triggered at judicious thresholds, thereby averting OOM errors and enhancing application resilience. This adaptive methodology enables Node.js applications to operate reliably within stringent memory confines and scales gracefully to clustered deployments. Nonetheless, it is essential to recognize that V8 flag tuning is a complementary optimization; it does not obviate the necessity for rigorous application-level memory profiling and remediation of latent memory leaks. The insights presented herein underscore the enduring importance of a holistic approach, one that synthesizes runtime configuration, virtual machine heuristics, and infrastructure awareness, to realize robust, performant, and cloud-native software systems.

REFERENCES

[1] Node.js Documentation Team. (2025). Understanding and Tuning Memory. Node.js Foundation.
[2] V8 Development Team. (2025). V8 JavaScript Engine Documentation. Google.
[3] Heroku Dev Center Team. (2024). Troubleshooting Node.js Memory Use. Heroku Dev Center.

[4] Node.js Documentation Team. (2025). V8 API Documentation. Node.js Foundation.
[5] Spencer, T. (2016). Running a Node.js app in a low-memory environment. Personal Blog.
[6] Egorov, V., & Corry, E. (2011). A game changer for interactive performance. Chromium Blog.
[7] V8 Team. (2021). Optimizing V8 memory consumption. V8 Official Blog.
[8] StrongLoop Team. (2015). How-to: Heap Snapshots and Handling Node.js Memory Leaks. StrongLoop Blog.
[9] V8 Team. (2019). A lighter V8. V8 Official Blog.
[10] Sasidharan, D. K. (2020). Visualizing memory management in V8 Engine (JavaScript, NodeJS, Deno, WebAssembly). Technical Blog.
[11] Stack Overflow Community. (2013). Configuring v8's memory management to be smart for a node.js process. Stack Overflow.
[12] V8 Users Community. (2013). How to use v8::Persistent::SetWeak to avoid memory leak. V8 Users Google Group.
[13] Node.js Contributors. (2019). Workaround for V8 design that leads to memory leak? Node.js GitHub Issue #25846.
[14] V8 Team. (2016). Trash talk: the Orinoco garbage collector. V8 Official Blog.
[15] Shestak, I. (2018). Garbage collection in V8, an illustrated guide. Medium.
[16] Binieli, M. (2025). Garbage Collector in V8 Engine. Medium.
[17] Conrod, J. (2013). A tour of V8: Garbage Collection. jayconrod.com.
[18] Ji, J. (2021). Memory Management in V8, garbage collection and improvements. DEV Community.
[19] Krylov, G., et al. (2020). The Evolution of Garbage Collection in V8: Google's JavaScript Engine. ResearchGate.
[20] Dijkstra, E. W. (1978). On-the-Fly Garbage Collection: An Exercise in Cooperation. Communications of the ACM, 21(11), 966-975.
[21] Printezis, T., & Detlefs, D. (2000). A Generational Mostly-Concurrent Garbage Collector. Sun Microsystems Technical Report.
[22] Levanoni, Y., & Petrank, E. (2001). An on-the-fly Reference Counting Garbage Collector for Java. Proceedings of OOPSLA.
[23] Appel, A. W. (1989). Simple Generational Garbage Collection and Fast Allocation. Software Practice and Experience, 19(2), 171-183.
[24] Wilson, P. R. (1992). Uniprocessor Garbage Collection Techniques. Proceedings of IWMM.
[25] Jones, R., & Lins, R. (1996). Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons.
[26] Mozilla Developer Network. (2025). Memory management - JavaScript. MDN Web Docs.
[27] Daily Team. (2023). Introduction to memory management in Node.js applications. Daily Blog.
[28] Imsushant, S. (2024). Advanced Memory Management and Garbage Collection in Node.js. DEV Community.
[29] Singh, A., et al. (2022). Memory Management with Nodejs. Codedamn News.
[30] Node.js Documentation Team. (2025). Memory. Node.js Official Documentation.
[31] Amoled, A. (2021). Memory Management with Node.js. DEV Community.
[32] Chen, S., et al. (2024). Cost Optimization in Cloud Computing: Capacity Reservation for Intermittent Random Demand Surges. Production and Operations Management, 33(6), 1265-1284.
[33] Alyatama, A. (2018). Memory allocation algorithm for cloud services. The Journal of Supercomputing, 74, 1035-1054.
[34] Gomathi, B., & Karthikeyan, K. (2013). Task scheduling algorithm based on hybrid particle swarm optimization in cloud computing. Applied Information Technology, 55, 33-38.
[35] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735-1780.
[36] Khan, T., et al. (2022). Machine Learning (ML)-Centric Resource Management in Cloud Computing: A Review and Future Directions. arXiv preprintarXiv:2105.05079.
[37] Waibel, P., et al. (2017). Cost-optimized redundant data storage in the cloud. Service Oriented Computing and Applications, 11(4), 411-426.
[38] Wang, C., et al. (2017). Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. Proceedings of EuroSys, 620-634.
[39] Wang, H., et al. (2021). GeoCol: A geo-distributed cloud storage system with low cost and latency using reinforcement learning. IEEE Transactions on Parallel and Distributed Systems, 32(2), 281-294.

[40] Syed, S., et al. (2024). Optimizing Cloud Resource Allocation with Machine Learning: A Comprehensive Approach to Efficiency and Performance. ResearchGate.