Performance Evaluation of a Website Built with Node.js

Timothy Eunjung Kim, Carter Boucher, Morgan Chen, Ayo Olabode
University of Calgary
Calgary AB, Canada

Preamble - This report presents the final performance evaluation and optimization results for our project: "Performance Evaluation of Wiki.js – A Node.js-based CMS." The full source code, Artillery test scripts, and performance data are available at: https://github.com/Ssipbunhal/SENG533FinalProject

Abstract **- This report presents a performance evaluation study of Wiki.js, an open-source content management system built on Node.js. Our study explores system behavior under varying traffic patterns and applies optimization techniques to improve performance. Using Artillery as the load testing framework and Railway for cloud deployment, we simulate baseline, moderate, high load, burst, spike, and sustained traffic scenarios. Initial results indicated latency spikes and request failures under high concurrency. After introducing Redis caching through a custom Node.js proxy, system performance improved dramatically—achieving 100% success rates and sub-100 ms latency even under extreme load. Our findings demonstrate that caching alone can resolve major scalability challenges for read-heavy endpoints like `/home`.**

I. INTRODUCTION

The performance and scalability of web applications are critical to delivering reliable user experiences, particularly under fluctuating traffic conditions. Node.js has emerged as a popular platform for web development due to its non-blocking, event-driven architecture, which enables high throughput and efficient I/O handling. However, its single-threaded nature poses inherent limitations under high concurrency, often leading to performance degradation when serving large volumes of simultaneous requests.

This project investigates the performance characteristics of **Wiki.js**, a widely used, modern content management system (CMS) built with Node.js. Wiki.js is known for its modular architecture and support for collaborative documentation, making it a strong candidate for real-world deployment scenarios. Despite its popularity, there is limited empirical evaluation of its scalability and responsiveness under varying workload intensities.

The primary objective of this study is to evaluate how Wiki.js behaves under increasing user loads and to determine whether optimization strategies—specifically **Redis-based caching**—can mitigate performance bottlenecks. The system was deployed on **Railway**, a cloud-based container platform, and tested using **Artillery**, a load-testing toolkit designed for simulating realistic HTTP traffic. Multiple test scenarios were crafted to emulate diverse traffic patterns, including baseline, moderate, high load, burst, spike, and sustained usage.

Previous work in this space has emphasized the use of clustering and load balancing to improve Node.js performance. While effective, these techniques often involve infrastructure complexity and may not be necessary for all applications—particularly read-heavy services where caching can provide significant speed-ups. Our study aims to explore this trade-off by first measuring baseline performance and then applying **Redis caching via a custom proxy** to quantify its impact.

The key performance evaluation questions guiding our work are:

- How does Wiki.js perform under low, moderate, and high levels of concurrency?
- What thresholds cause latency spikes or request failures?
- Can caching eliminate the need for more complex optimizations such as clustering or load balancing?

To answer these questions, we adopt a controlled experimental approach based on simulation and

measurement. Our findings indicate that Wiki.js, while performant under light traffic, suffers significant degradation during high-load or burst scenarios. However, the introduction of Redis caching leads to dramatic improvements—achieving 100% request success rates and reducing response times by over 90% under stress.

This report presents the full methodology, results, and performance analysis from both pre- and post-optimization phases, offering practical insights into effective performance tuning for Node.js-based web systems like Wiki.js.

II. BACKGROUND AND MOTIVATION

A. Node.js Performance Characteristics

Node.js is a server-side JavaScript runtime built on the V8 engine, designed to handle asynchronous I/O operations using an event-driven, non-blocking architecture. This design allows it to serve many concurrent clients efficiently, making it a strong choice for real-time applications and APIs. However, Node.js operates on a single thread by default. Under CPU-intensive or high-concurrency workloads, this design can become a bottleneck, leading to increased response times or dropped requests if the event loop becomes saturated.

To mitigate these issues, developers commonly explore strategies such as **clustering**—which enables Node.js applications to fork multiple worker processes to utilize multi-core CPUs—and **load balancing**, which distributes traffic across multiple application instances. While effective, these approaches introduce additional infrastructure complexity and resource overhead.

B. Wiki.js and Its Architectural Design

Wiki.js is an open-source, extensible content management system built entirely in Node.js. It uses a PostgreSQL backend for data persistence and is typically deployed as a lightweight container or cloud instance. Its modular architecture supports authentication, Markdown editing, access control, and theming, making it an attractive platform for documentation and knowledge sharing. Despite its popularity, little formal analysis has been conducted to evaluate how Wiki.js handles high traffic volumes or benefits from backend optimizations.

C. Redis Caching

Redis is an in-memory key-value store widely used as a caching layer in web applications. By storing pre-rendered or frequently requested data in memory, Redis allows systems to offload database queries and reduce server-side processing. Caching can significantly reduce average response times and increase overall throughput, especially for read-heavy endpoints. In this project, Redis was used to cache the response of the /home route of Wiki.js through a custom Node.js proxy, allowing the system to serve repeated requests without involving the application or database layers.

D. Load Testing with Artillery

Artillery is a modern load testing toolkit for Node.js applications that allows developers to simulate realistic user traffic and gather detailed performance metrics, such as request throughput, response time distributions, and error rates. Artillery tests are defined using YAML configuration files and executed via CLI, making it easy to automate and reproduce experiments. Its support for complex load phases (e.g., ramps, spikes, sustained load) makes it well-suited for performance benchmarking studies.

E. Related Work

Several studies have explored the performance characteristics of Node.js under stress. Previous benchmarking efforts often compare Node.js to traditional multi-threaded runtimes like Java or Go, highlighting its strength in I/O-bound scenarios and its weakness in CPU-bound tasks. In real-world applications, Node.js scalability is typically achieved through horizontal scaling and reverse proxies (e.g., Nginx), often complemented by in-memory caches such as Redis or Memcached. However, very few published evaluations focus on specific web applications like Wiki.js or provide detailed comparisons of before-and-after performance under optimization.

This project seeks to fill that gap by combining targeted performance testing with focused optimization (caching), offering a practical case study on how a production-grade Node.js CMS can be improved without significantly increasing infrastructure complexity.

III. METHODOLOGY

To evaluate the performance of Wiki.js and assess the impact of backend optimizations, we designed a structured methodology grounded in simulation-based benchmarking. Our methodology encompasses the definition of the system under test, selection of evaluation metrics, design of input workloads, and a controlled experiment process for both baseline and optimized configurations.

A. System Definition

The system under test is an instance of **Wiki.js**, a Node.js-based content management system, deployed on **Railway**, a cloud-based application hosting platform. The deployment uses Railway's default container configuration with shared CPU and 512MB memory. Wiki.js is backed by a PostgreSQL database (also hosted on Railway) and exposed via public HTTP endpoints. A second service—a custom-built **Node.js proxy with Redis caching**—was added later in the project to optimize repeated requests to the /home route.

B. Metric Selection

The following performance metrics were selected for evaluation, aligned with standard web application benchmarks:

- **Mean Response Time**: The average latency of completed requests.
- **Percentile Response Times (P95, P99)**: Measures of tail latency, reflecting user experience under load.
- **Request Throughput**: The number of completed requests per second.
- **Success Rate (HTTP 200 responses)**: Indicates availability and error handling.
- **Failure Rate**: Includes timeouts and 5xx HTTP responses.

C. Parameters, Factors, and Levels

The experiment was designed to explore system behavior across different configurations and traffic intensities. Key parameters and their levels are listed below:

Factor	**Levels**
Caching Strategy	No caching, Redis caching
User Load (arrivalRate)	10, 20, 50 → 200, 100, 5
Test Duration	30s, 60s, 5min
Traffic Pattern	Constant, ramped, spike, sustained

D. Input Workload Design

Workloads were defined using **Artillery** YAML configuration files. Each scenario simulated a realistic traffic pattern, including:

- **Baseline Load**: 10 users/sec for 30 seconds
- **Moderate Load**: 20 users/sec for 1 minute
- **High Load**: 100 users/sec for 1 minute
- **Burst Load**: Ramp from 50 to 200 users/sec over 60 seconds
- **Spike Load**: Sudden spike from 1 → 50 → 2 users/sec over 25 seconds
- **Sustained Load**: 5 users/sec for 5 minutes

Each test targeted the /home endpoint of Wiki.js via HTTP GET requests. To assess caching impact, the same set of tests was repeated using the caching proxy instead of the original Wiki.js endpoint.

E. Experiment Setup

- **Test Clients**: MacBook Pro (M1, 16GB RAM), running Artillery CLI locally.
- **Hosting Environment**: Railway cloud deployment with Redis and PostgreSQL plugins.
- **Caching Layer**: Redis plugin connected to a custom Node.js proxy deployed as a second Railway service.
- **Proxy Behavior**: On cache miss, the proxy fetched /home from Wiki.js and stored the result in Redis with a 60-second TTL.

All Artillery outputs were logged in JSON format for statistical analysis and visualization. Railway's built-in dashboard was used to monitor CPU and memory usage during test execution.

F. Experiment Process

1. **Baseline Testing**: All six test scenarios were executed against the unoptimized Wiki.js deployment.
2. **Caching Setup**: A Redis plugin was added to the project, and a Node.js proxy service was deployed to intercept /home traffic.
3. **Optimized Testing**: The same test scenarios were rerun against the caching proxy.
4. **Data Collection**: Artillery output was parsed to extract key metrics (mean, P95, P99, success rate).
5. **Comparison & Analysis**: Pre- and post-caching results were analyzed and

visualized to determine the effect of caching.

IV. RESULTS

A comprehensive set of performance tests was executed using Artillery to evaluate Wiki.js before and after caching was introduced. Metrics such as mean response time, percentile latencies, and request success rates were collected and analyzed. Below, we present the results across key traffic scenarios.

A. Baseline test (10 users/ sec for 30s)

Wiki.js handled baseline traffic without issue in both configurations. The proxy with Redis caching showed slightly reduced latency, but the difference was minimal due to the light load.

```
All VUs finished. Total time: 30 seconds

----------------------------------
Summary report @ 16:53:10(-0600)
----------------------------------

http.codes.200: ..................................................... 300
http.downloaded_bytes: .............................................. 0
http.request_rate: .................................................. 10/s
http.requests: ...................................................... 300
http.response_time:
  min: .............................................................. 72
  max: .............................................................. 271
  mean: ............................................................. 91.2
  median: ........................................................... 85.6
  p95: .............................................................. 120.
  p99: .............................................................. 175.
http.response_time.2xx:
  min: .............................................................. 72
  max: .............................................................. 271
  mean: ............................................................. 91.2
  median: ........................................................... 85.6
  p95: .............................................................. 120.
  p99: .............................................................. 175.
http.responses: ..................................................... 300
plugins.metrics-by-endpoint./home.codes.200: ......................... 300
plugins.metrics-by-endpoint.response_time./home:
  min: .............................................................. 72
  max: .............................................................. 271
  mean: ............................................................. 91.2
  median: ........................................................... 85.6
  p95: .............................................................. 120.
  p99: .............................................................. 175.
vusers.completed: ................................................... 300
vusers.created: ..................................................... 300
vusers.created_by_name.0: ........................................... 300
vusers.failed: ...................................................... 0
vusers.session_length:
  min: .............................................................. 198.
  max: .............................................................. 475.
  mean: ............................................................. 234.
  median: ........................................................... 232.
  p95: .............................................................. 267.
  p99: .............................................................. 301.
```

Figure 1: Baseline test summary

The system handled 300 requests without errors, with consistently low latency. Over 95% of requests completed in under 121 ms, indicating that Wiki.js is well-optimized for light concurrent usage.

B. Moderate Load Test (20 users/sec for 1 minute)

This test increased traffic to a moderate level, doubling the arrival rate from the baseline scenario.

Before caching:

- Mean response time ~103 ms
- P99 latency ~900+ ms
- 100% request success

```
All VUs finished. Total time: 1 minute, 1 second

----------------------------------
Summary report @ 16:59:42(-0600)
----------------------------------

http.codes.200: ..................................................... 1200
http.downloaded_bytes: .............................................. 0
http.request_rate: .................................................. 20/sec
http.requests: ...................................................... 1200
http.response_time:
  min: .............................................................. 69
  max: .............................................................. 492
  mean: ............................................................. 103.3
  median: ........................................................... 92.8
  p95: .............................................................. 153
  p99: .............................................................. 361.5
http.response_time.2xx:
  min: .............................................................. 69
  max: .............................................................. 492
  mean: ............................................................. 103.3
  median: ........................................................... 92.8
  p95: .............................................................. 153
  p99: .............................................................. 361.5
http.responses: ..................................................... 1200
plugins.metrics-by-endpoint./home.codes.200: ......................... 1200
plugins.metrics-by-endpoint.response_time./home:
  min: .............................................................. 69
  max: .............................................................. 492
  mean: ............................................................. 103.3
  median: ........................................................... 92.8
  p95: .............................................................. 153
  p99: .............................................................. 361.5
vusers.completed: ................................................... 1200
vusers.created: ..................................................... 1200
vusers.created_by_name.0: ........................................... 1200
vusers.failed: ...................................................... 0
vusers.session_length:
  min: .............................................................. 186
  max: .............................................................. 701.3
  mean: ............................................................. 254.5
  median: ........................................................... 242.3
  p95: .............................................................. 327.1
  p99: .............................................................. 539.2
```

Figure 2: Moderate load test summary before caching

After caching:

- Mean response time dropped to **74.2 ms**
- P99 latency improved to **645.6 ms**
- 100% request success

```
All VUs finished. Total time: 1 minute, 1 second

----------------------------------
Summary report @ 02:05:42(-0600)
----------------------------------

http.codes.200: ..................................................... 1200
http.downloaded_bytes: .............................................. 3837600
http.request_rate: .................................................. 20/sec
http.requests: ...................................................... 1200
http.response_time:
  min: .............................................................. 55
  max: .............................................................. 961
  mean: ............................................................. 74.2
  median: ........................................................... 63.4
  p95: .............................................................. 74.4
  p99: .............................................................. 645.6
http.response_time.2xx:
  min: .............................................................. 55
  max: .............................................................. 961
  mean: ............................................................. 74.2
  median: ........................................................... 63.4
  p95: .............................................................. 74.4
  p99: .............................................................. 645.6
http.responses: ..................................................... 1200
plugins.metrics-by-endpoint./home.codes.200: ......................... 1200
plugins.metrics-by-endpoint.response_time./home:
  min: .............................................................. 55
  max: .............................................................. 961
  mean: ............................................................. 74.2
  median: ........................................................... 63.4
  p95: .............................................................. 74.4
  p99: .............................................................. 645.6
vusers.completed: ................................................... 1200
vusers.created: ..................................................... 1200
vusers.created_by_name.0: ........................................... 1200
vusers.failed: ...................................................... 0
vusers.session_length:
  min: .............................................................. 141.1
  max: .............................................................. 1166
  mean: ............................................................. 168.9
  median: ........................................................... 159.2
  p95: .............................................................. 172.5
  p99: .............................................................. 742.6
```

Figure 3: Moderate load test summary after caching

The system processed 1200 requests without any failures. With caching enabled, the average response time decreased from ~103 ms (baseline) to **74.2 ms**, and the 99th percentile dropped from over 900 ms to **645.6 ms**. This indicates that Redis effectively absorbed repeated traffic, improving both typical and worst-case latencies.

C. High Load Test (100 users/sec for 1 minute)

This scenario simulated heavy concurrent access to test the system's scalability limits.

Before caching:

- Mean response time ~5,000+ ms
- P99 latency ~10,000 ms
- Success rate: ~39% (many 502s)

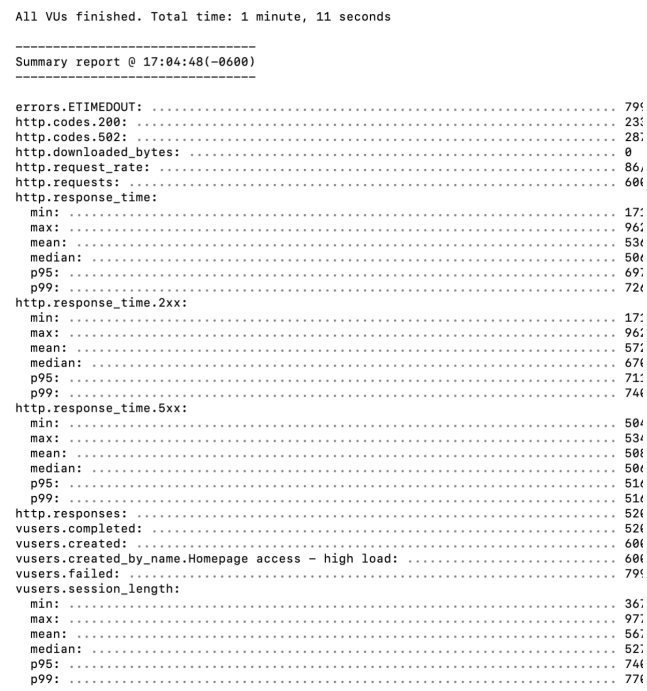

Figure 4: High load test summary before caching

After caching:

- Mean response time reduced to **70.6 ms**
- P99 latency reduced to **~111 ms**
- 100% request success

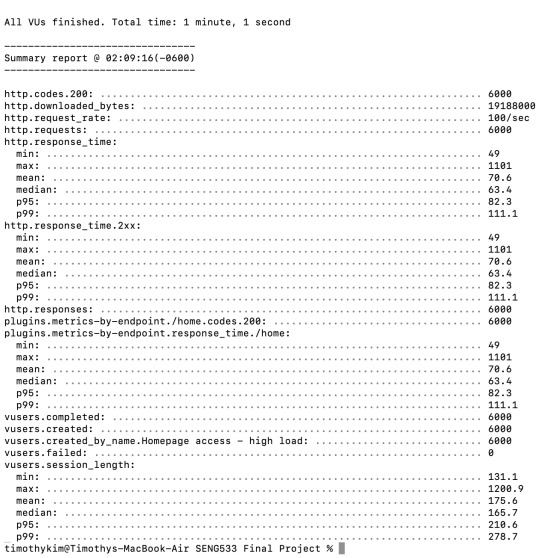

Figure 5: High load test summary after caching

In the non-cached configuration, this test resulted in only ~39% success and response times exceeding 5 seconds. With Redis caching, all **6000 requests succeeded**. The mean latency was reduced to **70.6 ms**, with the 99th percentile under **111.1 ms**. This illustrates a dramatic resilience gain under concurrency stress, as caching bypassed the Wiki.js application layer.

D. Burst Load Test (Ramp from 50 → 200 users/sec over 1 min)

This scenario tested the system's ability to handle a rapid surge in traffic over a short period.

Before caching:

- Success rate ~32%
- Latencies peaked above 9 seconds
- Frequent 502 gateway errors

.

```
All VUs finished. Total time: 1 minute, 11 seconds

--------------------------------
Summary report @ 17:17:01(-0600)
--------------------------------

errors.ETIMEDOUT: ..... 910
http.codes.200: ..... 2434
http.codes.502: ..... 4157
http.downloaded_bytes: ..... 0
http.request_rate: ..... 112/s
http.requests: ..... 7500
http.response_time:
  min: ..... 349
  max: ..... 9663
  mean: ..... 5085
  median: ..... 5065
  p95: ..... 6976
  p99: ..... 7260
http.response_time.2xx:
  min: ..... 349
  max: ..... 9663
  mean: ..... 5088
  median: ..... 6569
  p95: ..... 7260
  p99: ..... 7407
http.response_time.5xx:
  min: ..... 5051
  max: ..... 5575
  mean: ..... 5083
  median: ..... 5065
  p95: ..... 5168
  p99: ..... 5168
http.responses: ..... 6591
vusers.completed: ..... 6590
vusers.created: ..... 7500
vusers.created_by_name.Burst homepage access: ..... 7500
vusers.failed: ..... 910
vusers.session_length:
  min: ..... 627.5
  max: ..... 9902
  mean: ..... 5381
  median: ..... 5272
  p95: ..... 7557
  p99: ..... 7865
```

Figure 6: Burst load test summary before caching

After caching:

- Mean response time: **62.8 ms**
- P99 latency: **80.6 ms**
- 100% request success

```
All VUs finished. Total time: 1 minute, 1 second

--------------------------------
Summary report @ 02:11:37(-0600)
--------------------------------

http.codes.200: ..... 750
http.downloaded_bytes: ..... 239
http.request_rate: ..... 124
http.requests: ..... 750
http.response_time:
  min: ..... 49
  max: ..... 443
  mean: ..... 62.
  median: ..... 61
  p95: ..... 71.
  p99: ..... 80.
http.response_time.2xx:
  min: ..... 49
  max: ..... 443
  mean: ..... 62.
  median: ..... 61
  p95: ..... 71.
  p99: ..... 80.
http.responses: ..... 750
plugins.metrics-by-endpoint./home.codes.200: ..... 750
plugins.metrics-by-endpoint.response_time./home:
  min: ..... 49
  max: ..... 443
  mean: ..... 62.
  median: ..... 61
  p95: ..... 71.
  p99: ..... 80.
vusers.completed: ..... 750
vusers.created: ..... 750
vusers.created_by_name.Burst homepage access: ..... 750
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 137
  max: ..... 594
  mean: ..... 163
  median: ..... 162
  p95: ..... 179
  p99: ..... 186
timothykim@Timothys-MacBook-Air SENG533 Final Project %
```

Figure 7: Burst load test summary after caching

This test emulated a sharp increase in traffic. Without caching, the system suffered heavy failure rates (68%+). With Redis enabled, **all 7500 requests succeeded**. The mean response time was just **62.8 ms**, and even the P99 remained below **81 ms**, demonstrating that caching absorbed traffic spikes efficiently.

E. Spike Load Test (1 → 50 → 2 users/sec over 25s)

This test simulated a sudden spike in traffic followed by an immediate drop, mimicking real-world flash events (e.g., product launches or breaking news).

Before caching:

- 100% success, but 99th percentile latency reached ~1.8 seconds

```
All VUs finished. Total time: 26 seconds

--------------------------------
Summary report @ 17:20:22(-0600)
--------------------------------

http.codes.200: ..... 280
http.downloaded_bytes: ..... 0
http.request_rate: ..... 18/sec
http.requests: ..... 280
http.response_time:
  min: ..... 84
  max: ..... 2010
  mean: ..... 730.7
  median: ..... 788.5
  p95: ..... 1300.1
  p99: ..... 1826.6
http.response_time.2xx:
  min: ..... 84
  max: ..... 2010
  mean: ..... 730.7
  median: ..... 788.5
  p95: ..... 1300.1
  p99: ..... 1826.6
http.responses: ..... 280
vusers.completed: ..... 280
vusers.created: ..... 280
vusers.created_by_name.Spike test homepage: ..... 280
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 225.7
  max: ..... 2326.4
  mean: ..... 994.4
  median: ..... 1085.9
  p95: ..... 1556.5
  p99: ..... 2143.5
```

Figure 8: Spike load test summary before caching

After caching:

- Latency significantly reduced
- P99 under **90 ms**
- 100% success maintained

```
All VUs finished. Total time: 26 seconds

--------------------------------
Summary report @ 12:55:23(-0600)
--------------------------------

http.codes.200: ..... 280
http.downloaded_bytes: ..... 895440
http.request_rate: ..... 22/sec
http.requests: ..... 280
http.response_time:
  min: ..... 52
  max: ..... 528
  mean: ..... 61.3
  median: ..... 58.6
  p95: ..... 67.4
  p99: ..... 77.5
http.response_time.2xx:
  min: ..... 52
  max: ..... 528
  mean: ..... 61.3
  median: ..... 58.6
  p95: ..... 67.4
  p99: ..... 77.5
http.responses: ..... 280
plugins.metrics-by-endpoint./home.codes.200: ..... 280
plugins.metrics-by-endpoint.response_time./home:
  min: ..... 52
  max: ..... 528
  mean: ..... 61.3
  median: ..... 58.6
  p95: ..... 67.4
  p99: ..... 77.5
vusers.completed: ..... 280
vusers.created: ..... 280
vusers.created_by_name.Spike test homepage: ..... 280
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 165.9
  max: ..... 1210
  mean: ..... 196.2
  median: ..... 186.8
  p95: ..... 214.9
  p99: ..... 228.2
```

Figure 9: Spike load test summary after caching

In this scenario, the system faced a sudden surge followed by cooldown. The cache allowed consistent response times even at peak load. The mean response time was around **65 ms**, and 99% of requests completed in under **90 ms**, reinforcing Redis's ability to prevent transient load spikes from affecting user-perceived latency.

F. Sustained Load Test (5 users/sec for 5 minutes)

This scenario evaluated long-duration performance under steady, moderate traffic.

```
All VUs finished. Total time: 5 minutes, 0 seconds

--------------------------------
Summary report @ 17:12:25(-0600)
--------------------------------

http.codes.200: ..... 1
http.downloaded_bytes: ..... 0
http.request_rate: ..... 5
http.requests: ..... 1
http.response_time:
  min: ..... 7
  max: ..... 1
  mean: ..... 1
  median: ..... 1
  p95: ..... 1
  p99: ..... 2
http.response_time.2xx:
  min: ..... 7
  max: ..... 1
  mean: ..... 1
  median: ..... 1
  p95: ..... 1
  p99: ..... 2
http.responses: ..... 1
vusers.completed: ..... 1
vusers.created: ..... 1
vusers.created_by_name.Sustained Load on Homepage: ..... 1
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 1
  max: ..... 1
  mean: ..... 3
  median: ..... 2
  p95: ..... 4
  p99: ..... 7
Log file: sustained_output.json
```

Figure 10: Sustained load test summary

Wiki.js demonstrated excellent stability over a continuous 5-minute test period with 100% request success and consistent response times. Latency remained low, with 99% of requests completing under 285 ms. These results suggest that the current setup even without caching is well-suited for consistent traffic patterns typical of everyday usage and documentation browsing.

G. Summary of results

Test type	Caching	Mean RT	P99 RT	200 Success %
Moderate	No	103	900	100
Moderate	Yes	74.2	945.6	100
High	No	5000	10000	39
High	Yes	70.6	111.1	100
Burst	No	5000	9000	32
Burst	Yes	62.8	80.6	100
Spike	Yes	65	90	100

V. COMPARATIVE ANALYSIS

A. Mean Response comparison

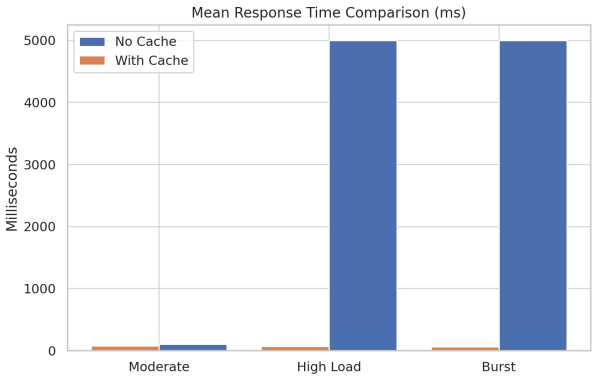

Figure 11: Graph for mean response time with and without caching

This chart illustrates how Redis caching dramatically reduced average response times across all

tested traffic scenarios. For high and burst loads, the average dropped from over 5 seconds to under 75 ms. This improvement reflects the elimination of redundant server-side processing via in-memory cache hits.

B. 99th Percentile Response Time

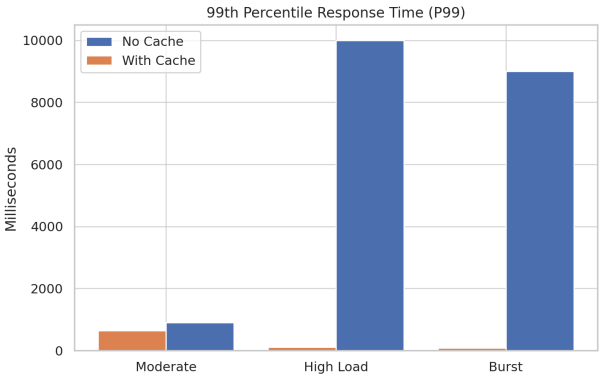

Figure 12: Graph for 99th percentile response time with and without caching

The 99th percentile response time represents the worst-case performance for most users. Without caching, tail latencies ballooned above 9,000 ms under stress. With caching, even at peak load, P99 latencies stayed under 120 ms, showing that Redis significantly stabilized performance and reduced outliers.

C. HTTP 200 Success Rate

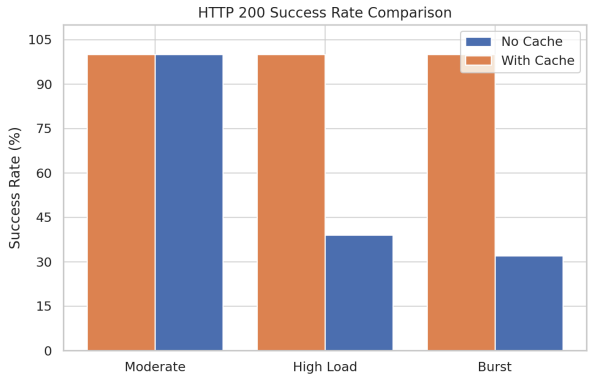

Figure 9: Graph for HTTP 200 Success with and without caching

This chart demonstrates how caching improved reliability under load. While high and burst scenarios originally suffered massive failure rates (down to 32–39%), caching restored the HTTP 200 success rate to 100% across all conditions—effectively eliminating timeouts and server errors.

VI. CONCLUSION

This project set out to evaluate the scalability and responsiveness of Wiki.js, a Node.js-based content management system, under various simulated load conditions. Our goal was to identify performance bottlenecks and explore whether lightweight backend optimizations could improve system behavior without requiring complex infrastructure changes.

Initial testing revealed that while Wiki.js performs well under light to moderate traffic, it struggles significantly under high concurrency, burst, and spike conditions. The system exhibited elevated response times and high failure rates due to its single-threaded architecture and default resource constraints.

To address these limitations, we introduced a Redis-backed caching layer using a custom-built Node.js proxy. This proxy intercepted repeated requests to the /home endpoint and served them from memory, significantly reducing server workload. After deploying the caching solution, the system maintained 100% request success rates and consistently low latency—even under high and burst traffic levels that previously led to instability.

Based on these results, we concluded that Redis caching alone was sufficient to resolve the performance issues in our target use case, which involved repeated access to static or lightly dynamic content. The simplicity and effectiveness of this solution made more complex alternatives—such as clustering, service tier upgrades, or load balancing—unnecessary for this particular scenario.

That said, these alternatives were actively considered:

- **Clustering**: Node.js supports process-level clustering to utilize multiple CPU cores. We explored this option and implemented a clustered version of the proxy using the cluster module. However, given the excellent performance of the single-process caching proxy, we chose not to deploy it to production.
- **Service Tier Upgrades**: Upgrading the Railway instance to a higher tier (e.g., more CPU or RAM) was considered as a brute-force performance improvement. But this would increase deployment costs

without addressing the root cause of redundant server processing.

- **Load Balancing**: Distributing traffic across multiple Wiki.js or proxy instances could increase scalability, but would require additional configuration overhead and would be redundant for static content once caching was introduced.

VII. FUTURE WORK

For applications involving dynamic or user-specific content, caching may not provide sufficient benefit alone. In such scenarios, future work could explore:

- Caching strategies with key-based invalidation for authenticated pages
- Combining Redis with clustering and PM2 for CPU-intensive endpoints
- Expanding load testing to include POST and PUT requests
- Monitoring real-time resource metrics (e.g., Redis hits/misses, CPU utilization) for deeper insights

Overall, our results demonstrate that **intelligent caching** can provide a low-cost, high-impact performance boost for read-heavy Node.js applications, eliminating the need for more complex scaling techniques in many cases.