การปรับปรุงอัตราการส่งคำขอเอพีไอของแพลตฟอร์มจัดการบทความด้วยหน่วยความจำแคชบนแพลตฟอร์ม CMS แบบไม่มีส่วนหัว
Improvement of API Request Throughput of Article Platform with Cache Memory on Headless CMS Platform

ชวลิต โควีระวงศ์[1] ณัฐรดี อนุพงค์[1] วสิศ ลิ้มประเสริฐ[2] และ ดาวรถา วีระพันธ์[1*]
Chavalit Koweerawong[1] Natradee Anupong[1] Wasit Limprasert[2] and Daorathar Weerapan[1*]
[1]คณะวิทยาศาสตร์และเทคโนโลยี มหาวิทยาลัยราชภัฏวไลยอลงกรณ์ ในพระบรมราชูปถัมภ์ จังหวัดปทุมธานี
[2]วิทยาลัยสหวิทยาการ มหาวิทยาลัยธรรมศาสตร์ จังหวัดปทุมธานี
[1]Faculty of Science and Technology, Valaya Alongkorn Rajabhat University under Royal Patronage, Pathum Thani Province
[2]College of Interdisciplinary Studies, Thammasat University, Pathum Thani Province
[*]Corresponding Author: daorathar@vru.ac.th

Received: March 27, 2025 **Revised:** June 1, 2025 **Accepted:** June 5, 2025

บทคัดย่อ

การศึกษานี้ศึกษาผลกระทบของเทคนิคการแคชต่อปริมาณงานและเวลาตอบสนองของ RESTful API ในสถาปัตยกรรม CMS แบบไม่มีส่วนหัวที่สร้างด้วย PHP Laravel ที่ทำงานร่วมกับ CMS ภายนอก ผู้ใช้ปลายทางมักประสบปัญหาในการใช้งานระบบ เช่น ความล่าช้าในการโหลดหน้าเว็บ ข้อมูลที่ไม่อัปเดตแบบทันที และข้อผิดพลาดระหว่างการใช้งาน ซึ่งปัจจัยเหล่านี้ส่งผลให้ประสบการณ์ผู้ใช้ลดลง และบั่นทอนความน่าเชื่อถือของระบบในภาพรวม วัตถุประสงค์หลักคือการปรับปรุงเวลาตอบสนองของ API ระหว่างเลเยอร์การนำเสนอและ CMS และเพื่อประเมินเทคนิคการแคชที่มีประสิทธิภาพสูงสุดสำหรับการเพิ่มประสิทธิภาพ API ของ Laravel ไดรเวอร์การแคชต่าง ๆ รวมถึงแคชที่ใช้ไฟล์ MySQL และ Redis ได้รับการทดสอบเพื่อวัดผลกระทบต่อปริมาณงาน API แคชทั้งสามแบบมีข้อดีและเสียที่แตกต่างกันโดยว่าไฟล์ และ Redis เป็นกลไกแคชในตัว ในขณะที่ MySQL เป็นฐานข้อมูลที่ดัดแปลงมาเพื่อจุดประสงค์ในการแคช WRK ถูกใช้เพื่อจำลองการใช้งาน API ในสถานการณ์สมจริง โดยสร้างคำขอผู้ใช้พร้อมกันไปยัง API การทดสอบแต่ละครั้งเกี่ยวข้องกับการส่งคำขอไปยังแอปพลิเคชันโดยมีผู้ใช้พร้อมกันเพียงรายเดียวต่อเธรดเป็นเวลา 60 วินาที โดยใช้ขนาดเพย์โหลดที่แตกต่างกันคือ 20KB, 200KB และ 2000KB ผลลัพธ์แสดงให้เห็นว่าการแคชช่วยปรับปรุงประสิทธิภาพของ API ได้อย่างมีนัยสำคัญ โดยการแคชที่ใช้ไฟล์มีปริมาณงาน 126.82 การร้องขอต่อวินาที เมื่อเทียบกับ 5.55 การร้องขอต่อวินาทีโดยไม่ใช้แคชที่ขนาดข้อมูล 20 กิโลไบต์ในบรรดาไดรเวอร์แคชทั้งสามตัวที่ทดสอบ การแคชที่ใช้ไฟล์ให้การปรับปรุงประสิทธิภาพสูงสุด โดยเพิ่มขึ้น 20 เท่าเมื่อเทียบกับการไม่แคช อย่างไรก็ตามการแคชที่ใช้ไฟล์แม้ว่าจะง่ายต่อการตั้งค่ามีข้อจำกัดในด้านการปรับขนาดและความยืดหยุ่นในการปรับใช้ทำให้เหมาะกับงานขนาดเล็ก ในขณะที่ Redis เป็นทางเลือกที่ดีกว่าสำหรับแอปพลิเคชันประสิทธิภาพสูงเนื่องจากความสามารถในการปรับขนาดในแนวนอน ในทางกลับกันไม่แนะนำให้ใช้แคช MySQL เนื่องจากมีประสิทธิภาพต่ำกว่า ยกเว้นในกรณีที่โหลดแคชต่ำซึ่งตัวเลือกแคชอื่น ๆ ไม่สามารถใช้งานได้จริง

คำสำคัญ: เอพีไอ ซีเอ็มเอสแบบไม่มีส่วนหัว การแคช การเพิ่มประสิทธิภาพการทำงาน ปริมาณงาน

Abstract

This study examines the impact of caching techniques on the throughput and response time of RESTful APIs in a headless CMS architecture developed with PHP Laravel that connected with external CMS. End users frequently experience issues such as slow page loading, delayed content updates, and occasional system errors. These factors contribute to a diminished user experience and undermine the overall reliability of the system. The primary objectives are to enhance the API response time between the Presentation Layer and the CMS and to evaluate the most effective caching technique for optimizing Laravel's API performance. Various caching drivers, including file-based caching, MySQL caching, and Redis caching, were tested to assess their effects on API throughput. All three caches have their own advantages and disadvantages, with File and Redis being built-in cache mechanisms, while MySQL is a database adapted for caching purposes. WRK was used to simulate realistic API usage by generating concurrent user requests to the API. Each test involved sending requests to the application with a single concurrent user per thread for 60 seconds, using varying payload sizes of 20KB, 200KB, and 2000KB. The results indicate that caching significantly improves API performance, with file-based caching achieving a throughput of 126.82 requests per second compared to 5.55 requests per second without caching at payload 20 KB. Among the three cache drivers tested, file-based caching provided the greatest performance improvement, with 20 times increase compared to no caching. However, file-based caching while easy to config, has limitations in terms of scalability and deployment flexibility, making it more suitable for small applications. Redis was found to be a superior alternative for high-performance applications due to its horizontal scalability. In contrast, MySQL caching was not recommended due to its relatively low performance, except in scenarios of low cache loads where other caching options were impractical.

Keywords: API, Headless CMS, Caching, Performance optimization, Throughput

1. Introduction

A Headless CMS (Content Management System) is a content management system that provides content without a front-end website. It allows developers to build websites and applications that consume and display content as needed. This approach decouples the content from the presentation layer, giving developers with greater flexibility to create engaging user experiences. Kamal (2022) suggests that a Content Management System (CMS) is a cost-effective option, as it enables individuals without coding knowledge to develop websites in traditional CMS systems where the front-end and back-end are tightly coupled. However, in a Headless CMS, the back-end is entirely decoupled from the front-end, resulting in greater flexibility and scalability. In the research context, a headless CMS offers a key benefit: write content once and display it across multiple subdomains within the organization. Typically, a Headless CMS requires an API (Application Programming Interface) to allow other applications or websites to retrieve and display content. In a Headless CMS, content is stored separately from the presentation layer, meaning the CMS does not dictate how the content is presented. Instead, the content is made available through an API, allowing developers to access and use it in their own applications or websites. The API usually contains content in a structured format, such as JSON or XML, that can be easily consumed by other applications. This empowers developers to create customized front-end applications or websites that can display content in a manner that suits their specific needs, without being concerned about the underlying CMS or presentation layer. Integrating APIs with external CMS in a headless architecture often encounters challenges such as slow response times and instability due to external dependencies. Data retrieval delays can negatively impact user experience, while caching limitations, rate limiting, and API version changes pose further complications. Debugging becomes more complex, requiring in- depth understanding of interconnected systems. These issues collectively reduce the overall performance and reliability of the platform.

One of the most critical factors influencing Web API performance is response time. Response time refers to the duration it takes for the API to respond to a client's request. Slow response times can lead to a degraded user experience, resulting in high bounce rates and negatively impacting the overall API performance. According to the study by Babovic et al. (2016), response time is a key determinant of the overall efficiency of a web API. Reducing response time can significantly enhance Web API efficiency, resulting in improved user experience and increased user engagement.

To improve API speed and efficiency, several key strategies can be implemented. Caching plays a crucial role by storing frequently accessed data in memory using tools like file, database, Redis, significantly reducing response times. JSON compression techniques such as GZIP help minimize payload size, enabling faster data transmission over the network. Integrating a Content Delivery Network (CDN) allows static API responses and media files to be served from edge locations closer to users, decreasing latency. Moreover, load balancing evenly allocates incoming requests across multiple servers, helping to prevent system overload and maintain stable performance during periods of high traffic. Combined with other optimization strategies, this approach supports the development of an API infrastructure that is scalable, efficient, and resilient (Bautista et al., 2023; Vemasani & Modi, 2024).

Caching is a widely used technique for improving the performance of web APIs by temporarily storing frequently accessed data, allowing subsequent requests to be served more quickly without repeatedly querying the underlying data source. Various research studies have investigated and validated the effectiveness of caching strategies in optimizing API performance across a range of platforms and application scenarios (Mertz & Nunes, 2017; Hasnain et al., 2020; Sudda, 2024; Shi et al., 2024). They found that this approach greatly reduces response latency, leading to faster load times and more efficient resource utilization.

Laravel Cache is a powerful caching library that can be used to implement caching in Laravel-based web APIs. Consistent with the work of Ahmed et al. (2024), that focused on a comparative analysis of performance optimization techniques applied to two PHP frameworks, Laravel and CodeIgniter, in implemented various optimization strategies, including caching, query optimization, and code refactoring, and measured their impact on the frameworks' response times over single and multiple iterations. Laravel Cache is a popular caching library for PHP-based web applications. This library provides a simple and efficient approach to cache data and reduce web API response times. Laravel Cache can be utilized to cache database queries and API responses, and it seamlessly integrated into Laravel-based web applications. There are many caching techniques like file, Redis, and database caching, each with different performance benefits. Laravel supports multiple cache drivers, making it easy to improve performance by caching data, queries, or full API responses efficiently without additional hardware.

A RESTful API (Representational State Transfer API) is a web service that follows the principles of REST (Representational State Transfer) architecture (Rodríguez et al, 2016) to enable communication between clients and servers over the internet. It is widely used for building scalable, maintainable, and stateless web applications. It enhances scalability, simplicity, and performance by enforcing statelessness and resource-oriented interactions. This architectural style has become the foundation of modern RESTful APIs, significantly influencing web standards, particularly in HTTP-based API design (Fielding, 2000). By leveraging the advanced capabilities of the Laravel framework, the proposed approach seeks to optimize the development process of RESTful APIs, thereby improving efficiency and maintainability. Chen et al. (2017) emphasize Laravel's built-in functionalities, which ensure compliance with REST architectural principles, facilitating the design of scalable and resilient web services. This study provides significant insights into the adoption of Laravel for RESTful API development, contributing to the establishment of best practices in modern web service architecture.

Therefore, this study aims to improve the response time of a RESTful API between the presentation layer and a headless content management system, and to evaluate the most effective caching techniques in PHP Laravel for optimizing API throughput and response times.

2. Methodology

2.1 Experimental Setup

This experiment is conducted to gather information about various types of caching in the PHP Laravel framework, their advantages and disadvantages, as well as cache functionalities. This case study involves implementing caching in a Laravel application and measuring the impact on performance and throughput relevant to APIs and caching used for Headless CMS. Throughput is the rate of work that can be done. In the case of Web APIs, it is the rate of processing API calls, specifically the average number of requests that can be successfully processed per second or per minute over a given period.

Laravel offers a standardized API for multiple caching backends, enabling seamless transitions between different caching systems without requiring modifications to the application code. The framework supports various caching drivers, including Array, File, Database, Memcached, Redis, DynamoDB, and Octane. However, this experiment focuses on File Cache, Database Cache, and Redis Cache.

File Cache stores cached data as individual files in the filesystem. While simple to implement, it suffers from slower read/write performance compared to Redis due to disk I/O limitations. Additionally, managing large cache files can become inefficient in high-traffic applications.

Database Cache mechanism stores cache data in a database table, making it easier to query and manage. However, retrieving cached data requires database queries, introducing latency and reducing performance under heavy load.

Redis Cache operates entirely in-memory, resulting in ultra-fast read and write operations. It also supports advanced caching features like automatic expiration, data persistence, and pub/sub messaging. Due to its ability to scale horizontally, Redis is the preferred choice for applications requiring high concurrency and low-latency responses. Redis can sometimes be slower by its communication over the network, even on localhost.

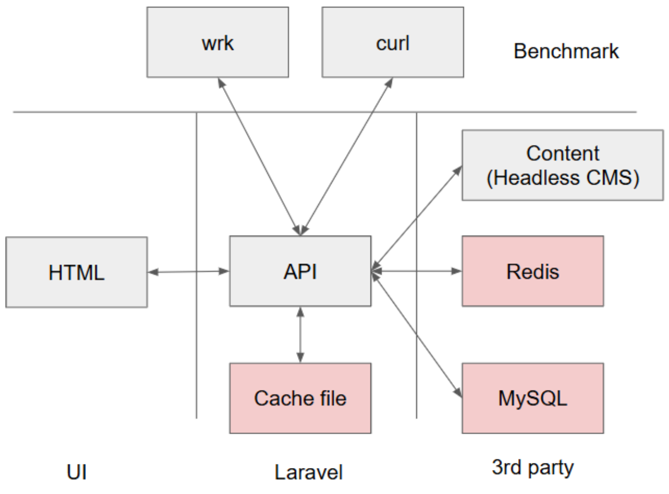

Figure 1. Components used in the experiment

WRK is a modern open-source HTTP benchmarking tool that enables the measurement of the performance and throughput of web servers or API servers. It is designed to generate a high load on web applications to simulate realistic scenarios and evaluate how well a server handles the load. To measure the throughput of cached responses, the ‘curl’ command was used to execute PHP scripts that record the time taken to retrieve data using the ‘get cache’ command, successfully fetching cached data across different caching types.

The experiment was conducted to evaluate the impact of different caching techniques on the performance and throughput of a RESTful API. The API application was developed using the Laravel framework (version 10) and connected to a MySQL database and a Redis server. The application was deployed on a cloud server with 1 vCPU, 1GB RAM, and a 25GB SSD, running a LEMP stack (Linux/Ubuntu 22, NGINX, MySQL, PHP 8.2). Additionally, Redis Server and WRK were installed for performance benchmarking.

2.2 Testing Procedure

To measure the effect of caching on API performance, a structured testing procedure was implemented. The experiment aimed to evaluate various caching mechanisms under controlled conditions to assess their impact on response time, throughput, and resource utilization.

2.2.1 Load Simulation

WRK was used to simulate realistic API usage by generating concurrent user requests to the API. Each test involved sending requests to the application with a single concurrent user per thread for 60 seconds, using varying payload sizes of 20KB, 200KB, and 2000KB.

Table 1. Experimental Environment

Configurations	Sizes
Concurrent user	1, 100, 200 concurrent user(s)
CPU	1 thread
Duration	60 seconds
Payload Size	20KB, 200KB, 2000KB
Cache Size	1 – 1,000 records

2.2.2 Caching Implementation

Laravel's built-in caching system was utilized to implement and compare various caching methods. The evaluation was conducted under three distinct test conditions: File Cache, where responses were stored in Laravel's file-based caching system; MySQL Database Cache, where cached data was stored in a MySQL database table; and Redis Cache, an in-memory caching system, which was employed for storing cached responses.

2.2.3 Performance Metrics

To comprehensively evaluate the caching mechanisms, this study measured the following performance metrics, which are often used in various research studies (Ahmed et al., 2024; Arifiany & Kusuma, 2025) to assess the effectiveness of caching strategies: Response Time (ms), which refers to the duration taken for the API to respond to each request; Throughput (requests per second), the number of successful API requests processed per second; Throughput vs. Payload Size, which analyzes how different caching mechanisms handled varying payload sizes in terms of processing efficiency; and CPU and Memory Usage, where the system resource consumption for each caching method was recorded to determine the trade-offs between performance and resource utilization (Babovic et al., 2016) including/excluding API processing in the measurement process for testing power of only cache without API lag overhead.

2.3 Data Collection and Analysis

The response times and throughput metrics were systematically logged during each test. To ensure accuracy, multiple iterations of each test were conducted, and the collected data was analyzed using descriptive statistics (Kamal, 2022). In this experiment, each test was conducted 10 times, with throughput and response time reported as the mean and standard deviation of the observed values. The performance differences between caching mechanisms were visualized using bar and line charts. Throughput was compared across different payload sizes and cache sizes to determine which caching mechanism offered the best scalability and efficiency. The results were then interpreted to identify the most effective caching strategy for optimizing Laravel API performance in a Headless CMS environment.

2.4 Limitations and Considerations

Despite the structured methodology, certain limitations existed in the experimental setup. The experiment was conducted on a single-node server without horizontal scaling, which may limit its ability to handle higher traffic loads (Bhatt, 2024). Horizontal scaling may be considered for implementation in future work to improve scalability. Additionally, only three caching mechanisms (File, Redis, and MySQL) were evaluated, while other caching technologies such as Memcached and DynamoDB were not included in the study. Moreover, network latency and external traffic variations were not considered since the tests were performed in a controlled environment.

Measuring the impact of caching on performance and throughput, WRK was used to simulate user traffic on the application. The experimental scenario involved sending one concurrent user per thread to the application for 60 seconds, while using different payload sizes of 20KB, 200KB, and 2000KB. Throughput was monitored throughout the test. Cache drivers were implemented in the sample application using Laravel's built-in caching functions. The cache drivers tested included File, Redis, MySQL Database, and No Cache. The results were compared to evaluate the impact of caching on performance and throughput. The collected data was analyzed using descriptive statistics and visualized with bar and line charts.

3. Results

3.1 Installation Complexity

The complexity of installation and configuration varies significantly across different caching drivers, as summarized in Table 2., indicates that the File driver does not require any installation or configuration steps, while Redis and MySQL necessitate the installation of software and configuration within the application to connect. Redis on Windows may require additional setup for proper functionality.

These differences in installation complexity highlight that while file caching is the easiest to implement, it lacks scalability. Conversely, Redis and MySQL require additional setup but provide better support for distributed applications or horizontally scalable by network I/O configuration (Camilleri et al., 2024).

3.2 Throughput, Response Time vs. Payload Size

The performance evaluation measured the throughput of the API under different caching mechanisms using payload sizes of 20KB, 200KB, and 2000KB, simulating various realistic traffic scenarios in a headless CMS application. The results are summarized in Table 3. and Table 4.

Table 2. Installation Complexity of Each Caching Driver

Driver	Installation Condition
File	- No dependencies required, simply use the built-in file driver in the framework. - The cache storage location is in the folder /framework/cache/data. - Supported on all operating systems.
Redis	- Redis is a key-value store, which requires specific configuration for the connection. - Redis is officially supported on Windows using WSL2 (Windows Subsystem for Linux). The performance can be more efficient for running on Linux. - Requires installation and configuration within the application. - Network I/O configuration
MySQL	- MySQL Server needs to be installed with some php-mysql extension for PHP. - Create a database and some schemas specifically for caching. - Requires configuration with the application to ensure proper connectivity. - Network I/O configuration

Table 3. API Throughput vs. Payload Size

Driver	Throughput (request/second)					
	Payload 20KB		Payload 200KB		Payload 2000KB	
	Mean	SD	Mean	SD	Mean	SD
File	126.82	14.19	115.88	13.93	63.28	7.70
Redis	113.71	13.81	96.92	13.04	49.81	6.84
MySQL	104.31	12.98	85.28	10.82	48.76	6.44
No cache	5.55	1.99	3.66	0.88	2.45	0.31

Table 4. API Response time vs. Payload Size

Driver	Response Time (ms)					
	Payload 20KB		Payload 200KB		Payload 2000KB	
	Mean	SD	Mean	SD	Mean	SD
File	7.89	1.33	8.63	1.43	15.80	2.35
Redis	8.79	1.62	10.31	1.83	20.08	2.88
MySQL	9.59	1.65	11.73	1.57	20.51	2.43
No cache	180.18	10.59	273.22	14.03	408.16	49.99

File Cache outperforms other caching mechanisms across all payload sizes with the highest throughput. At 20KB, the file caching method demonstrated a throughput of 126.82 req/s, SD = 14.19, outperforming Redis, which achieved 113.71 req/s, SD = 13.81, and MySQL, which recorded 104.31 req/s, SD = 12.98. As the payload size increased to 2000KB, the File Cache remained the fastest but dropped to 63.28 req/s, SD = 7.70, showing a decline in performance due to file system limitations. Redis provided better performance than MySQL in all scenarios. At 200KB, Redis achieved 96.92 req/s, SD = 13.04, while MySQL lagged at 85.28 req/s, SD = 10.82. At 2000KB, Redis performance declined to 49.81 req/s, SD = 6.84, though it remained marginally more efficient than MySQL, which achieved 48.76 req/s, SD = 6.44. Both caching methods, relying on network communication, still suffered performance degradation at higher payload sizes. The API without caching (No cache) is the baseline, achieving 5.55 req/s, SD = 1.99, which is more than 20 times slower than any cache driver. As shown in Table 4., considering API response time alongside API throughput, the File Cache driver is the fastest in all payload sizes.

The results of the payload simulation scenario demonstrate that caching has a significant impact on the performance and response time of the sample Laravel application. File caching is best suited for small-scale applications where simplicity and fast access to local files are priorities. However, it lacks horizontal scalability. Redis offers a balance between performance and scalability, making it the preferred choice for large-scale applications requiring low-latency and high-throughput responses. MySQL caching is the least effective for high traffic or high payload applications and is primarily useful in cases where existing database infrastructure must be leveraged for caching. However, caution is needed to avoid system bottlenecks caused by database caching.

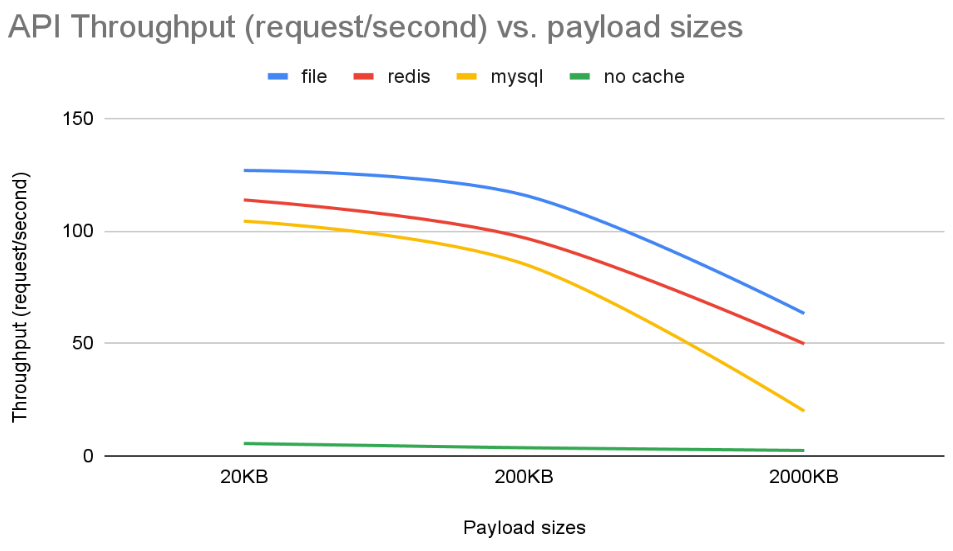

Figure 2. API Throughput vs. Payload Size

3.3 Cache Throughput vs. Payload Size

To analyze the caching mechanism alone (excluding API processing overhead), an additional benchmark was performed focusing purely on cache retrieval throughput. This test aimed to measure the effect of caching alone, where both the cache and no-cache scenarios should be in the same direction with different throughput results. File Cache again demonstrated the fastest response times, maintaining higher throughput across all payload sizes. At a payload size of 20KB, the File driver achieved a mean throughput of 25,006 req/s, SD = 3,034, which was 2.5 times higher than Redis (10,002 req/s, SD = 1,334), and over 31 times higher than MySQL (791 req/s, SD = 112). However, its performance depends on disk I/O speed, and for applications with limited storage, this could be a bottleneck. At the highest payload of 2000KB, throughput dropped sharply across the board: File at 164 req/s (SD = 22), Redis at 124 req/s (SD = 18), and MySQL at 98 req/s (SD = 12). Despite the sharp drop, File Cache maintained the lead, but the performance gap between File and Redis narrowed significantly. Redis performed better than MySQL, confirming its efficiency as an in-memory cache. However, Redis's advantage diminishes with large payloads due to the memory cost of storing large objects, which aligned with the study of Shi et al. (2024) regarding Redis's advantages in data structure support and persistence features, and highlights issues like scalability and multi-core CPU utilization. MySQL struggled with larger payloads, reaffirming that databases are not optimized for high-speed caching. While file caching provides excellent single-server performance, it is not suitable for distributed environments. Redis and MySQL, on the other hand, support remote access, clustering, and horizontal scaling, making them more appropriate for applications with growing traffic. Intuitively, memory operation (Redis) should be faster than storage (file), Redis can sometimes be slower than file-based caching in Laravel because it communicates over the network I/O, even on localhost, introducing slight latency due to TCP connections and data serialization. Moreover, on modern SSD with light traffic, file I/O can be faster than Redis in some situations with small application or low traffic.

Table 5. Cache Throughput vs. Payload Size

Driver	Throughput (request/second)					
	Payload 20KB		Payload 200KB		Payload 2000KB	
	Mean	SD	Mean	SD	Mean	SD
File	25,006	3,034	8,822	1,131	164	22
Redis	10,002	1,334	2,807	400	124	18
MySQL	791	112	539	64	98	12

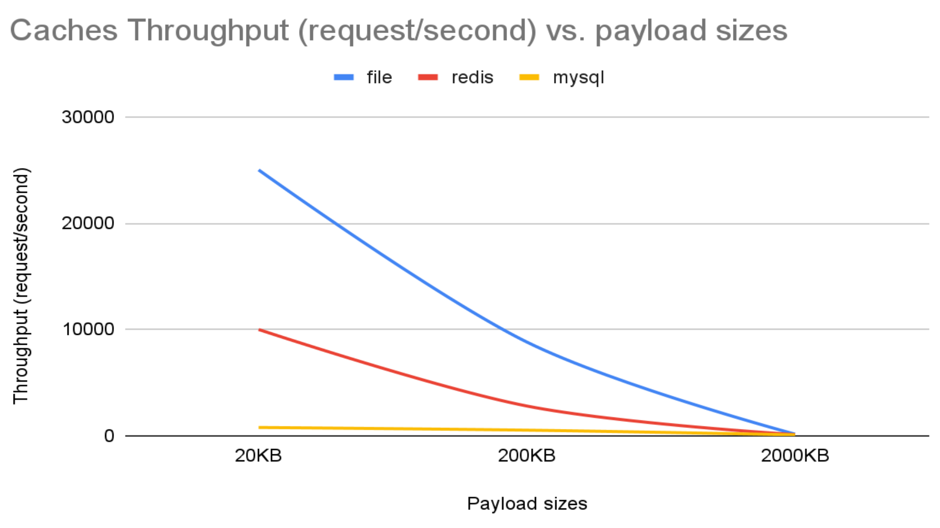

Figure 3. Cache Throughput vs. Payload Size

3.4 Cache Throughput vs. Cache Size

Cache throughput refers to how many read/write operations a cache can handle per second, while cache size determines how much data it can store. While a larger cache size can reduce cache misses by storing more data and tend to cause higher throughput as table 6, it doesn't always lead to higher throughput. At a cache size of 10 records, File driver achieved the highest throughput at 125.80 req/s (SD = 15.02), followed by MySQL at 97.59 req/s (SD = 13.13) and Redis at 97.24 req/s (SD = 15.08). As the cache size increased to 100 records, File throughput slightly declined to 116.71 req/s (SD = 16.71), Redis dropped to 87.49 req/s (SD = 27.41), while MySQL improved to 101.40 req/s (SD = 11.71). At 1,000 records, File throughput slightly recovered to 118.90 req/s (SD = 17.37), Redis increased to 102.21 req/s (SD = 15.46), and MySQL decreased to 90.90 req/s (SD = 13.79). These results show that Redis is more stable at higher cache sizes, while MySQL fluctuates and may degrade due to database read overhead. The File Cache maintains consistent performance across all sizes, though slightly impacted at 100 records.

In-memory caches like Redis offer high throughput with low latency but are limited by RAM, whereas file-based and MySQL cache may support larger sizes but have slower access speeds due to disk I/O. Optimizing API performance requires balancing both cache size and throughput to ensure fast responses and efficient data access.

3.5 Cache Requests vs. Concurrent users

The number of cache requests typically increases as concurrent users grow, each user request may trigger a cache read. With effective caching, most of these user requests are served directly from the cache, reducing database load and improving response times. However, if the cache isn't sized or configured properly, high concurrency can lead to higher timeout requests and degrading performance. To maintain stability, the cache system must scale to handle increasing read/write throughput as concurrent user counts rise.

Table 6. Cache Throughput vs. Cache Size with Payload 20KB

Driver	Throughput (request/second)					
	10 records		100 records		1,000 records	
	Mean	SD	Mean	SD	Mean	SD
File	125.80	15.02	116.71	16.71	118.90	17.37
Redis	97.24	15.08	87.49	27.41	102.21	15.46
MySQL	97.59	13.13	101.40	11.71	90.90	13.79

As shown in Table 7., with 100 concurrent users, the File Cache handled 6,596 requests with no timeouts, demonstrating excellent single-node efficiency. MySQL also performed well with 5,288 requests and no timeouts, indicating acceptable performance at low load due to direct database access. Redis, while faster in memory access, recorded 2,412 requests with 140 timeouts, showing stable behavior under low concurrency. When the concurrency doubled to 200 users, File Cache still performed relatively well with 6,656 requests and 638 timeouts. Redis's performance dropped further to 2,210 received requests and 1,177 timeouts, likely due to memory pressure or network I/O bottlenecks. MySQL showed a drastic increase in timeout behavior with 5,181 requests and 5,018 timeouts, meaning nearly all requests experienced latency or failure to respond within time limits.

These results highlight that while File and MySQL caches handle light concurrency well, they deteriorate rapidly under higher user loads. Redis, although designed for high-concurrency workloads, needs proper configuration and more system resources to meet expectations. Although MySQL is convenient in certain infrastructures, it is not optimized for caching under high concurrency and exhibited critical bottlenecks due to limitations in concurrent query execution and disk access contention.

Table 7. API Request vs. Concurrent Users in 1 Minute

Driver	Requests 20 KB			
	100 concurrent users		200 concurrent users	
	Received	Timeout	Received	Timeout
File	6,596	0	6,656	638
Redis	2,412	140	2,210	1,177
MySQL	5,288	0	5,181	5,018

3.6 Scalability and Infrastructure Considerations

Although the results demonstrate an impressive performance range for the file driver, it is important to note that the file driver is only suitable for single-node servers, where the API and cache reside in the same file storage. On the other hand, Redis and MySQL can be deployed on separate nodes and connected over the network. If insufficient file space is an issue, File Cache can easily fail. Additionally, for APIs that need to share cache data, Redis and MySQL are more appropriate solutions. Both Redis and MySQL are designed to scale both horizontally and vertically, whereas file caching is limited by the performance and capacity of the file system, resulting in lower scalability for large applications or high-traffic scenarios on cache of Laravel Framework. Although MySQL can function as a cache, it is not its primary purpose since MySQL is typically used for persistent storage and can become a bottleneck in routine query services. In rare cases, MySQL may be selected for caching simply due to existing infrastructure, but Redis remains a more efficient and scalable caching solution compared to MySQL.

The scalability of each caching mechanism was evaluated, as summarized in Table 8. This result may provide additional insights to inform decision-making regarding the selection of appropriate cache drivers for specific situations.

Table 8. Experiment Scenarios

Scenarios	File	Redis	MySQL
remote server	FALSE	TRUE	TRUE
cluster	FALSE	TRUE	TRUE
vertical scale	TRUE	TRUE	TRUE
horizontal scale	FALSE	TRUE	TRUE
replication	FALSE	TRUE	TRUE

Based on the experimental outcomes, File Cache, while efficient in localized deployments, showed significant limitations, especially when subjected to concurrency stress and increasing data sizes. As seen in Table 7, it experienced higher timeout rates when user concurrency exceeded 200, mainly due to its reliance on file I/O and lack of network-level scalability. Redis and MySQL support advanced infrastructure capabilities like clustering, remote access, and replication. However, Redis outperforms MySQL in read/write speed, memory efficiency, and cache-specific features, making it more appropriate for dynamic, high-throughput environments.

In conclusion, File Cache is ideal for simple, single-node setups with minimal scalability needs. MySQL may be used where integration is prioritized over performance, but Redis remains the optimal solution for scalable, distributed, and high-performance Laravel applications.

3.7 Advantages and Disadvantages of Each Cache

A comparative analysis of each caching mechanism is presented in Table 9.

Table 9. Comparison of the advantages and disadvantages of each cache

Cache Driver	Advantages	Disadvantages
File	- Easy to set up and use as it utilizes the file system for cache storage. - Suitable for small applications or development environments. - No need for additional dependencies or configurations.	- Performance may degrade as the cache size grows due to file system operations. - Limited scalability and performance compared to other drivers. No horizontal scaling. - May be slower than in-memory caches like Redis for certain workloads.
Redis	- Fast in-memory data caching with high performance and low latency. - Provides advanced caching features like expiration, tagging, and pub/sub messaging. - Network I/O provides Highly scalable, handling large datasets and high traffic loads, ideal for horizontal scaling. - Supports more complex data structures and operations beyond basic key-value caching.	- Requires a Redis server, adding extra dependency. - Needs specific setup and configuration for connecting to the Redis server. - More complex configuration compared to file or MySQL caching. - Network I/O Overhead made some certain lags even though enhance fast speed from in-memory.
MySQL	- Leverages the existing MySQL database infrastructure, requiring no additional setup. - Ideal for applications already using MySQL and wishing to use the same database for caching. - Provides data persistence and durability.	- Performance may degrade compared to file-based or in-memory caching due to disk I/O and database operations. - Can increase the workload on the MySQL server. - Limited scalability and performance compared to dedicated caching systems. Can scale horizontally but at a higher cost than Redis.

The comparison of caching mechanisms reveals distinct advantages and disadvantages for each. File caching is the simplest to implement, requiring no additional setup or dependencies, making it ideal for small-scale applications. However, it suffers from performance degradation as the cache size grows and lacks scalability. Redis caching offers fast in-memory data access, advanced features like expiration policies, and excellent scalability through clustering and replication, making it ideal for high-concurrency applications. The trade-off is that it requires more complex setup and consumes more memory. MySQL caching, while leveraging existing infrastructure and offering data persistence, is the least efficient for caching due to slower performance from database query overhead, making it better suited for persistent storage rather than real-time cache retrieval.

4. Discussion

In summary, the results of the case study demonstrate that caching has a significant impact on the performance and throughput of the Laravel API application. The File Cache driver provided slightly better performance compared to Redis and MySQL drivers by the result of throughput and response time, although with limitations when used on a single-node machine, which can only improve performance through vertical scaling. In contrast, both Redis and MySQL can be deployed anywhere and connected through a network layer, offering more flexibility in deployment. Using MySQL as a caching mechanism is considered a rare case due to performance concerns and potential bottlenecks. The result of cache performance with and without API are in the same direction, while the without API shows better throughput due to no network impact.

The selection of an appropriate cache driver (File, MySQL, or Redis) depends on various factors, including the application's context, complexity, performance requirements, scalability needs, and available resources. File Cache is easy to set up but may have limitations in performance, especially in terms of scalability, as the cache resides within the API project code. MySQL caching can be beneficial if MySQL is already in use and you want to reuse the same infrastructure. MySQL offers better scalability than file-based caching because MySQL services can be separated from the API, though scaling MySQL remains challenging and costly.

In comparison with existing research, this study confirms several key observations. Hasnain et al. (2020) emphasized caching's role in improving scalability for web services in CMS platforms. Similarly, this study found Redis to be the most suitable for scalable, high-performance deployments, while file caching was effective only in single-node environments with limited scalability. Mertz & Nunes (2017) provided a broad theoretical foundation on caching benefits, such as reduced latency and better resource utilization. This work complements their findings by presenting experimental evidence specific to Laravel APIs, particularly regarding performance across varying payload sizes. The observed performance order (File, Redis, MySQL) is consistent with Sudda (2024), who demonstrated that server-side caching in Laravel improves responsiveness. Sudda’s suggestions for future enhancements, such as prefetching and rate limiting, offer valuable directions to extend this research. Compared with Bang et al. (2024), which quantitatively showed Redis outperformed MySQL in data operations by several factors, this study supports the conclusion in the context of API caching, especially for large payload scenarios. Moreover, Ahmed et al. (2024) benchmarked Laravel and CodeIgniter and highlighted Laravel's flexible caching system, an insight directly reflected in this research's ability to switch between multiple cache drivers.

Redis, on the other hand, is highly efficient and offers advanced features, but requires additional configuration. Therefore, selecting the most suitable cache driver for an application should align with its specific needs. This aligns with the study by Bang et al. (2024), which demonstrated that Redis outperforms MySQL, requiring 5.84 times less time for data insertion, 6.61 times less for data retrieval, and 12.33 times less for data deletion. These results indicate that Redis is particularly advantageous in environments requiring high data processing and maintenance. Consequently, organizations and online service providers may opt for NoSQL databases such as Redis to enhance data management efficiency.

File caching can be faster than Redis in some cases because it avoids network overhead and accesses data directly from local storage, making it more efficient for simple, low-traffic applications. For a small usage application like the experiment configuration, File Cache is recommended to use.

As the number of concurrent users increases, cache requests also rise. A well-optimized cache reduces database load by quickly serving frequent data; however, under high concurrency, it may cause higher timeouts, so the cache size (i.e., how much data can be stored) must be carefully balanced. In-memory caches like Redis offer high throughput but limited storage capacity, while file-based and MySQL caches allow more storage at the cost of slower access speed. To maintain performance, systems must scale caching strategies to handle both increasing user load and efficient data storage.

This research focuses on improving API performance for headless CMS platforms. File-based caching is the most suitable choice for small-scale applications, considering payload sizes between 20 KB and 2000 KB, cache sizes of 10 to 100 records, and concurrent usage of around 100 users.

5. Conclusion

In the research context, headless CMS enhance a crucial benefit feature that write content one, display many subdomains in the organization and the bottleneck of these system is slow API. Caching is an effective method for improving throughput and enhancing the performance of web applications built using the Laravel framework. This study evaluated three caching mechanisms (File Cache, Redis, and MySQL) within a Laravel-based API environment for optimizing RESTful API throughput and response times of headless content management system by using system without cache as baseline. The result of cache performance with and without API are in the same direction, while the without API shows better throughput due to no network impact. File caching is the simplest and fastest for single-node applications but lacks scalability. Redis offers the best balance of speed, scalability, and flexibility, making it the ideal choice for large-scale applications. MySQL is the least efficient for caching, as it introduces additional query overhead and is better suited for persistent storage rather than high-speed cache retrieval. For production applications requiring high throughput, low latency, and scalability, Redis may be the recommended caching solution. However, for small-scale applications or development environments, file caching remains a viable option due to its simplicity and keeps the best performance as the experimental result. The study also revealed that the File Cache driver provided performance that was 20 times better than the no-cache driver. Among the three cache drivers, the file driver performed the best, although it had limitations in scalability and deployment flexibility. For easy installation and low-cost implementation, the file driver is recommended, despite its limited scalability. For high performance and cost-effectiveness, Redis with high horizontal scalability is recommended. MySQL is not recommended as a cache due to its lower performance, except in cases with light cache loads where it does not create system bottlenecks and using file and Redis caching would be inconvenient. File caching can be faster than Redis in some cases because it avoids network overhead and accesses data directly from local storage, making it more efficient for simple, low-traffic applications. For a small usage application like the experiment configuration, File Cache is recommended to use. Future research could include studying caching in scenarios with very high loads (more than one concurrent user and test more than 60 seconds) and horizontal scaling.

6. Acknowledgements

We would like to express our sincere gratitude to the Thailand Science Research and Innovation (TSRI) for providing research funding from the government budget for the fiscal year 2023. We would also like to extend our thanks to all individuals who have contributed valuable data, feedback, and suggestions, which have greatly supported the successful completion of this research.

7. References

Ahmed, MK., Bello, AH., Jauro, SS., & Dawaki, M. (2024). A comparative analysis of performance optimization techniques for benchmarking PHP frameworks: Laravel and CodeIgniter. Dutse Journal of Pure and Applied Sciences, 10(3c), 284–295.

Arifiany, I., & Kusuma, GP. (2025). Leveraging headless content management system as a service in a service-based architecture: Enhancing user experience and overcoming resource limitations for start-ups. Journal of Theoretical and Applied Information Technology, 103(7), 2991–3011.

Babovic, ZB., Protic, J., & Milutinovic, V. (2016). Web performance evaluation for internet of things applications. IEEE Access, (4), 6974-6992.

Bang, H., Kim, SH., & Jeon, S. (2024). Comparative Evaluation of Data Processing Performance between MySQL and Redis. Journal of Internet Computing and Services, 25(3), 35-41.

Bautista, PB., Comellas, J., & Urquiza-Aguiar, L. (2023). Evaluating scalability, resiliency, and load balancing in software-defined networking. Engineering Proceedings, 47(1), 16.

Bhatt, S. (2024). Best practices for designing scalable REST APIs in cloud environments. Journal of Sustainable Solutions, 1(4), 48–71.

Camilleri, C., Vella, JG., & Nezval, V. (2024). Horizontally Scalable Implementation of a Distributed DBMS Delivering Causal Consistency via the Actor Model. Electronics, 13(17), 3367.

Chen, X., Ji, Z., Fan, Y., & Zhan, Y. (2017). Restful API Architecture Based on Laravel Framework. Journal of Physics: Conference Series, 910(1), 1-6.

Fielding R. (2000). Architectural Styles and the Design of Network-based Software Architectures. Dissertation. California: University of California.

Hasnain, M., Pasha, MF., & Ghani, I. (2020). Drupal core 8 caching mechanism for scalability improvement of web services. Software Impacts, 3, 100014.

Kamal, J. (2022). Design and Development of a Headless Content Management System. International Research Journal of Engineering and Technology (IRJET), 9(7), 697-700.

Mertz, J., & Nunes, I. (2017). Understanding application-level caching in web applications: A comprehensive introduction and survey of state-of-the-art approaches. ACM Computing Surveys, 9(4), Article 39.

Rodríguez, C., Báez, M., Daniel, F., Casati, F., Trabucco, J. C., Canali, L., & Percannella, G. (2016). REST APIs: A large-scale analysis of compliance with principles and best practices. International Conference on Web Engineering, Lecture Notes in Computer Science, 9671, 21–39.

Shi, L., Qiao, H., Yang, C., Jiang, Y., Yu, K., & Chen, C. (2024). Research and application of distributed cache based on Redis. Journal of Software, 19(1), 1-8.

Sudda, PR. (2024). Optimizing PHP API calls with pagination and caching. Master’s thesis. USA: Michigan Technological University, Michigan.

Vemasani, P., & Modi, S. (2024). Optimizing cloud computing performance: How CDNs revolutionize global content delivery. Journal of Advanced Research Engineering and Technology (JARET), 3(1), 11–24.