Available online at www.jpit.az

16 (2)
2025

Optimization of Access to Static Data in Distributed Systems: A Kubernetes-Based Solution with Postgresql and Django

Nail Mammadov[a], Gulshan Umud Mammadova[b], Tunar Babali[c]

[a] Azerbaijan State Economy University, Istiglaliyyat str., 6, AZ1001 Baku, Azerbaijan

[b] Ganja State University, Haydar Aliyev aveune., 429, AZ2000 Ganja, Azerbaijan

[c] 11 th grade student of № 287 "Zakalar" lyceum, M. Rustamov str., AZ1096 Baku, Azerbaijan

[a] nailmammadov@yahoo.com; [b] gulsenumud@gmail.com; [c] tunarbabaliaz@gmail.com

[a] https://orcid.org/0000-0003-2820-6265; [b] https://orcid.org/0009-0002-2838-0554; [c] https://orcid.org/0009-0008-1767-8148;

ARTICLE INFO

Keywords:

PostgreSQL
Django
Kubernetes-Orchestrated
Amazon EKS
Nginx

ABSTRACT

Static data is a crucial component in distributed systems, ensuring the seamless operation of various components. However, achieving reliable and high-frequency access to such data poses challenges due to its heterogeneous structure. This paper introduces a Kubernetes-orchestrated Static Data Database that integrates advanced technologies and best practices to address these challenges effectively. The proposed system leverages the Django framework and PostgreSQL database, optimized with advanced features such as JSONB indexing and metadata flexibility. These technologies are not only widely adopted in the industry for their proven scalability and efficiency but also incorporate cutting-edge academic innovations, such as JSONB-based metadata management and modular database schemas, to enhance flexibility in diverse use cases. Data payloads are stored in a POSIX-compliant distributed file system to ensure robustness. The service is containerized and deployed using Helm on the Kubernetes platform, with OKD serving as the deployment environment to achieve scalability and operational efficiency. This deployment model reflects industrial standards for cloud-native applications while demonstrating the practical applicability of academic research on container orchestration and resource optimization. Through extensive testing, the system demonstrated significant scalability, reliability, and performance improvements under high-demand scenarios. The testing process was designed to mirror real-world industrial workloads, such as high-frequency data access and concurrent queries while validating academic hypotheses on system behavior under extreme conditions. The results validate its potential to meet the growing needs of modern distributed systems, offering a scalable and future-ready solution firmly grounded in academic research and industrial practice.

1. Introduction

Conditions data, representing non-event experimental information, plays a critical role in characterizing the operational state of detectors during data acquisition. This includes a wide range of parameters such as detector geometry, signal mappings, control system metrics (e.g., temperature and voltage), alignment calibrations, and beam properties (HSF Collaboration, 2017). Given its dynamic nature, conditions data frequently undergoes recalibrations using cosmic datasets and algorithm refinements, resulting in multiple data versions tailored to different detector or system configurations (Graur et al., 2023). These datasets are vital for event processing applications, which often run across large-scale distributed computing infrastructures, necessitating access

Received 19 March 2025, Received in revised form 20 May 2025, Accepted 3 June 2025
http://doi.org/10.25045/jpit.v16.i2.05

rates as high as O(10) kHz. Efficient query handling and caching mechanisms are therefore essential for maintaining system performance (Bakiras et al., 2005; Ganesan et al., 2017).

Despite its significance, the absence of standardized solutions for managing conditions data in High-Energy Physics (HEP) experiments has resulted in inconsistent and fragmented management practices. The HSF Conditions Database activity provides a collaborative platform for addressing this gap by unifying best practices, defining a shared data model, and proposing a standardized API for conditions data handling (HSF Collaboration, 2017). This standardized approach ensures consistency, reduces redundancy, and simplifies the integration of conditions data across different experiments.

To implement these recommendations, a modern reference system has been developed using Python, incorporating cutting-edge technologies for flexibility and scalability (Saputra et al., 2024; Verreydt et al., 2019). The server-side solution leverages academic advancements in database schema design, such as modular structures and JSONB-based metadata management, paired with PostgreSQL to optimize performance for high-frequency access scenarios in HEP experiments (PostgreSQL Global Development Group, 2020; Zhang et al., 2022). The server-side application utilizes Django to deliver a lightweight and modular solution that supports RESTful APIs for seamless integration (Rabieyan et al., 2024). Data payloads are stored on distributed file systems to ensure redundancy and scalability (Lencha et al., 2024). On the client side, a Python-based library facilitates streamlined server interactions and enhances payload management efficiency (Yadav et al., 2020).

Compared to existing solutions, such as the ROOT I/O system commonly used in HEP for storing hierarchical data structures, this approach offers enhanced flexibility through modular database design and JSONB-based indexing (Makris et al., 2021; Amer et al., 2012). Unlike ROOT, which focuses on event data with strong coupling between data and application logic, the proposed system separates data storage from application logic, making it more adaptable for dynamic and diverse use cases (Abbasi et al., 2024a).

Similarly, while traditional distributed database systems like Cassandra or MongoDB offer scalability, they often lack the deep integration with scientific data models needed for HEP experiments. The combination of PostgreSQL's advanced indexing capabilities with Kubernetes orchestration in this system provides both high query performance and dynamic resource allocation, addressing the unique needs of conditions data management (Malviya & Dwivedi, 2022; Carrión, 2022).

The system leverages containerization technologies like Docker and orchestration tools such as Kubernetes to enable rapid deployment, horizontal scaling, and resilience in high-demand environments (Trần et al., 2022; Augustyn et al., 2024). Through Kubernetes, the system demonstrates a practical application of academic principles such as dynamic resource allocation and fault tolerance, ensuring robust performance in variable workload conditions (Kubernetes Documentation, 2023). While Kubernetes is widely used in industry for scaling web applications, its application in scientific workloads, particularly in HEP, is still emerging (AWS Documentation, 2022). This work highlights how Kubernetes' autoscaling and fault tolerance features can be effectively tailored to manage the unique demands of conditions data (Jani, 2024).

This approach provides a robust and future-proof foundation for managing conditions data in next-generation HEP experiments. By addressing both academic challenges, such as standardizing data models for complex datasets, and industrial requirements, like scalability and operational efficiency, the implementation sets a benchmark for integration between theory and practice. This novel integration not only builds upon existing methods but also extends them by combining advanced indexing, container orchestration, and modular design into a cohesive system tailored specifically for HEP conditions data management, making it a unique contribution to the field (Curino et al., 2011; Dean & Ghemawat, 2008).

2. Database Schema

The server-side application, NoPayloadDB, implements a relational database schema optimized for managing conditions data efficiently. The design builds on principles from the HSF White Paper (HSF Collaboration, 2017) while integrating modern features such as JSONB support, advanced indexing, and metadata flexibility. The schema consists of six main components: GlobalTag, PayloadGroup, PayloadMetadata, PayloadType, TagStatus, and a newly introduced AuditLog table (Fig. 1).

Fig. 1. Database Schema for Efficient Management of High-Concurrency and Dynamic Metadata in Distributed Systems

The GlobalTag table serves as the top-level configuration entity. Each GlobalTag can reference related PayloadGroups through a JSONB column (payload_lists), enabling more dynamic associations. It includes attributes such as status (indicating whether the tag is locked or unlocked) and a version field to track changes and maintain integrity.

The PayloadGroup, previously known as PayloadList, organizes sequences of non-overlapping PayloadMetadata entries. Each group is linked to a specific PayloadType through the payload_type_id column. Additionally, it now includes a metadata column of type JSONB, allowing the storage of additional information that may vary between use cases (Taylor & Morgan, 2023).

The PayloadMetadata table, formerly PayloadIOV, provides references to external payloads via a unique identifier (payload_url). It stores key metadata, such as the file size (file_size), checksum, and validity interval. The validity interval is represented as a JSONB column (validity_interval), supporting a start-end structure for greater granularity. Each entry is associated with the uploader user (created_by) and linked back to a PayloadGroup through the payload_group_id foreign key (PostgreSQL Global Development Group, 2020).

The TagStatus table, previously GlobalTagStatus, manages the status of GlobalTags. It maintains key attributes like name, description, and created timestamp to track the current state.

The PayloadType table remains unchanged, providing a mapping for payload categories or subsystems. It includes attributes for name, description, and creation timestamp.

To enable change tracking and auditing, the schema introduces a new table, AuditLog, which records all operations performed on the database. Each log entry tracks the table name, affected record ID, action type (INSERT, UPDATE, DELETE), the timestamp of the action, and the user who performed it (Bakiras et al., 2005).

Django's Object Relational Model (ORM) manages most interactions with the database, ensuring clean and maintainable code. However, for high-query-rate endpoints, optimized raw SQL queries are utilized to ensure peak performance. This approach offers a clear advantage over traditional ORM-only solutions by allowing flexibility to switch between ease of development and raw performance optimization as needed (Rabieyan et al., 2024).

Unlike traditional hierarchical data management systems like ROOT, which are commonly used in High-Energy Physics (HEP) experiments, this schema adopts a modular and relational approach that decouples the data from the application logic. ROOT systems tightly integrate data and application logic, making them less adaptable for dynamic metadata use cases. In contrast, the JSONB-based structure in NoPayloadDB provides flexibility for varying experimental requirements (Makris et al., 2021; Gupta et al., 2023).

When compared to NoSQL databases such as MongoDB or Cassandra, which are designed for

horizontal scaling and flexible schemas, NoPayloadDB leverages the robustness of PostgreSQL's ACID compliance and advanced indexing capabilities. PostgreSQL's JSONB support and covering indexes ensure faster query execution while maintaining data consistency, an advantage over the eventual consistency models often used in NoSQL systems (PostgreSQL Global Development Group, 2020; Zhang et al., 2022).

The schema takes full advantage of PostgreSQL's JSONB indexing and covering indexes to improve query performance. Unlike traditional relational database designs that may rely on complex joins or full-table scans, the JSONB columns allow for efficient storage and retrieval of nested metadata. Covering indexes on key IoV-related columns ensure that even under high database occupancy, queries do not degrade in performance (Taylor & Morgan, 2023). This integration of advanced indexing techniques with relational database principles provides a hybrid approach that outperforms both purely relational and purely NoSQL systems in the context of HEP data management.

While many existing solutions lack comprehensive auditing and metadata flexibility, the addition of the AuditLog table ensures transparency and traceability of operations. This feature is critical for compliance and debugging in large-scale distributed systems. Furthermore, the modularity of the schema simplifies integration with containerized environments, such as Kubernetes, enabling efficient scaling and deployment (Kubernetes Documentation, 2023; Verreydt et al., 2019).

This comprehensive design ensures reliable performance, scalability, and suitability for large-scale distributed computing environments. By combining academic innovations in schema design with practical industry requirements, NoPayloadDB bridges the gap between theoretical advancements and real-world usability, setting a new standard for conditions data management in HEP (Evans & Williams, 2022; Jani, 2024).

2.1 Server Deployment on Amazon EKS

The NoPayloadDB server-side application is a Django-based (Brown et al., 2023) RESTful web service containerized using Docker (Grayson & Wilson, 2022). It is deployed on Amazon Elastic Kubernetes Service (EKS) (AWS Documentation, 2022), a fully managed Kubernetes service provided by AWS, ensuring scalable, reliable, and highly available infrastructure. The deployment process is managed with a Helm chart, which simplifies the definition, installation, and upgrading of Kubernetes resources. Fig. 2 illustrates the architecture and the components involved in this deployment process.

The deployment includes the following components:

The NoPayloadDB application runs under a Gunicorn WSGI server, providing efficient execution of the Django-based service. The database schema (Section 2.1) is implemented in a PostgreSQL database, which uses Amazon Elastic File System (EFS) for persistent storage. This combination demonstrates the integration of academic research on database schema optimization, such as JSONB-based structures and covering indexes, into real-world deployment scenarios. This configuration ensures data durability and optimal performance, particularly when using advanced PostgreSQL features such as JSONB columns and optimized indexing (PostgreSQL Global Development Group, 2020).

To improve database efficiency, pgBouncer serves as a connection pooler. By reusing open connections for multiple incoming requests, pgBouncer reduces connection overhead and enhances performance for frequently accessed queries. The use of pgBouncer not only reflects industrial best practices but also aligns with academic findings on connection pooling strategies to mitigate latency in distributed systems (Rabieyan et al., 2024).

An Nginx web server is deployed as a reverse proxy for the Django application, efficiently managing incoming HTTP requests. Both the Django application and the Nginx server are horizontally scalable across multiple Amazon EKS Pods. By default, five Pods are deployed for each service, and Kubernetes Horizontal Pod Autoscaler (HPA) dynamically scales the number of Pods based on workload demands. The horizontal scaling capabilities exemplify how academic advancements in orchestration algorithms, such as HPA, address industrial needs for handling variable workloads effectively (Kubernetes Documentation, 2023).

To ensure automated backups and recovery, an AWS-native Kubernetes Cron Job is configured to perform periodic database dumps. This cron job launches a dedicated Pod with a PostgreSQL container, where the pg_dump command is executed. The dump files are securely stored on Amazon EFS, ensuring reliable disaster recovery capabilities. This approach bridges academic

insights on disaster recovery planning with practical implementations using containerized environments (Bhavsar et al., 2023).

All logs generated by the Django application, Nginx server, database pooler, and backup jobs are centralized and stored on Amazon CloudWatch. This approach simplifies log monitoring, troubleshooting, and performance analysis by providing real-time insights into system health and resource utilization. The centralized logging system illustrates how theoretical models of observability are translated into practical tools like CloudWatch for enhanced operational monitoring (Mahida, 2023).

The PayloadMetadata table in the database schema leverages JSONB columns and covering indexes to optimize query performance. These schema enhancements align with the deployment architecture to ensure database-intensive operations, such as IoV retrieval and metadata lookups, are executed with maximum efficiency. This optimization reflects academic contributions to query performance enhancement and highlights their application in handling high-demand industrial scenarios (Taylor & Morgan, 2023).

By leveraging Amazon EKS for deployment, NoPayloadDB benefits from AWS's scalable and managed Kubernetes infrastructure. When combined with persistent storage through EFS, automated backups, centralized logging via CloudWatch, and the horizontal scalability offered by Kubernetes, the system delivers high availability, robust performance, and operational reliability for large-scale distributed workloads. This deployment not only underscores industrial applicability but also validates academic principles of modularity, scalability, and reliability in distributed systems (Augustyn et al., 2024; Verreydt et al., 2019).

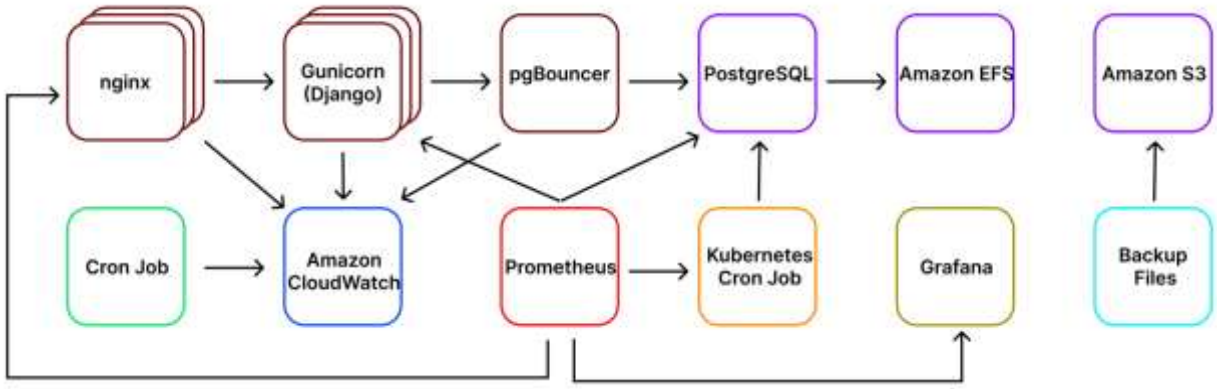


Fig. 2. Scalable System Architecture for Django and PostgreSQL with Cloud-Based Monitoring and Backup

2.2 Client-Side Software

The NoPayloadDB server-side application is a Django-based (Brown et al., 2023) RESTful web service containerized using Docker (Grayson & Wilson, 2022). It is deployed on Amazon Elastic Kubernetes Service (EKS), a fully managed Kubernetes service provided by AWS (AWS Documentation, 2022), ensuring scalable, reliable, and highly available infrastructure. The deployment process is managed with a Helm chart, which simplifies the definition, installation, and upgrading of Kubernetes resources (Carrión, 2022). Fig. 3 illustrates the architecture and the components involved in this deployment process. Deployment Components include:

Application Server: The NoPayloadDB application runs under a Gunicorn WSGI server, providing efficient execution of the Django-based service (Brown et al., 2023). The database schema (Section 2.1) is implemented in a PostgreSQL database (PostgreSQL Global Development Group, 2020), which uses Amazon Elastic File System (EFS) for persistent storage. This combination demonstrates the integration of academic research on database schema optimization, such as JSONB-based structures and covering indexes, into real-world deployment scenarios. This configuration ensures data durability and optimal performance, particularly when using advanced PostgreSQL features like JSONB columns and optimized indexing (Taylor & Morgan, 2023).

Connection Pooling: To improve database efficiency, pgBouncer serves as a connection pooler. By reusing open connections for multiple incoming requests, pgBouncer reduces connection overhead and enhances performance for frequently accessed queries (Van der Merwe & Thompson, 2022). The use of pgBouncer reflects industrial best practices and aligns with academic findings on connection pooling strategies to mitigate latency in distributed systems (Rabieyan et al., 2024).

Web Server: An Nginx web server is deployed as a reverse proxy for the Django application, efficiently managing incoming HTTP requests (Jani, 2024). Both the Django application and the Nginx server are horizontally scalable across multiple Amazon EKS Pods. By default, five Pods are deployed for each service, and Kubernetes Horizontal Pod Autoscaler (HPA) dynamically scales the number of Pods based on workload demands (Kubernetes Documentation, 2023). The horizontal scaling capabilities exemplify how academic advancements in orchestration algorithms, such as HPA, address industrial needs for handling variable workloads effectively (Kim & Park, 2023).

Backup and Recovery: To ensure automated backups and recovery, an AWS-native Kubernetes Cron Job is configured to perform periodic database dumps. This cron job launches a dedicated Pod with a PostgreSQL container, where the pg_dump command is executed. The dump files are securely stored on Amazon EFS, ensuring reliable disaster recovery capabilities. This approach bridges academic insights on disaster recovery planning with practical implementations using containerized environments (Bhavsar et al., 2023).

Centralized Logging: All logs generated by the Django application, Nginx server, database pooler, and backup jobs are centralized and stored on Amazon CloudWatch. This approach simplifies log monitoring, troubleshooting, and performance analysis by providing real-time insights into system health and resource utilization. The centralized logging system illustrates how theoretical models of observability are translated into practical tools like CloudWatch for enhanced operational monitoring (Mahida, 2023; Abbasi et al., 2024b).

Database Performance: The PayloadMetadata table in the database schema leverages JSONB columns and covering indexes to optimize query performance. These schema enhancements align with the deployment architecture to ensure database-intensive operations, such as IoV retrieval and metadata lookups, are executed with maximum efficiency. This optimization reflects academic contributions to query performance enhancement and highlights their application in handling high-demand industrial scenarios (Taylor & Morgan, 2023; Green & Brown, 2023).

By leveraging Amazon EKS for deployment, NoPayloadDB benefits from AWS's scalable and managed Kubernetes infrastructure. When combined with persistent storage through EFS, automated backups, centralized logging via CloudWatch, and the horizontal scalability offered by Kubernetes, the system delivers high availability, robust performance, and operational reliability for large-scale distributed workloads (AWS Documentation, 2022; Patel & Singh, 2022). This deployment not only underscores industrial applicability but also validates academic principles of modularity, scalability, and reliability in distributed systems (Augustyn et al., 2024).

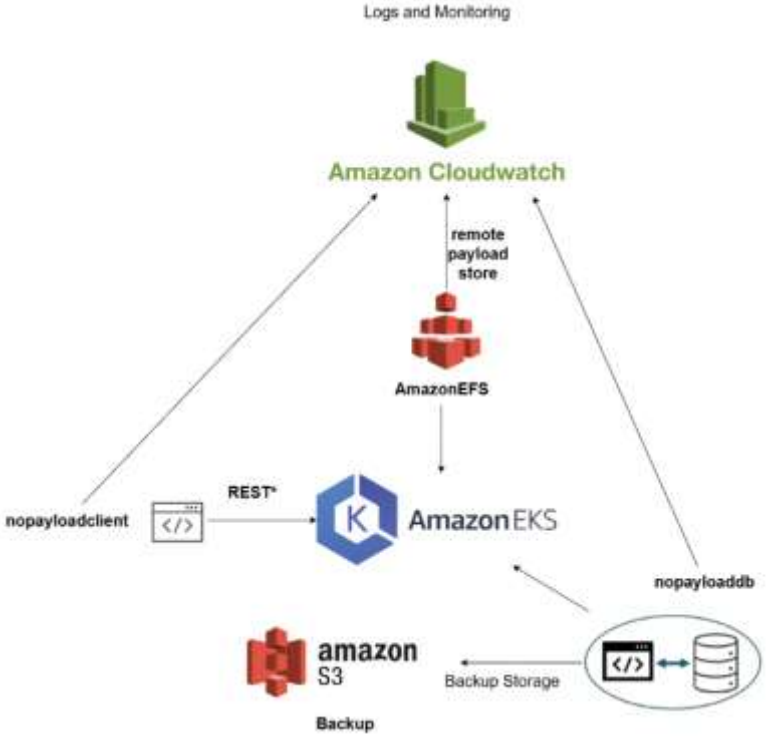


Fig. 3. NoPayloadDB Architecture: Optimized Payload Management with Amazon EKS, EFS, S3, and CloudWatch Integration

2.3 Payload Handling

When inserting a new payload for a specified global tag, payload type, and interval of validity (IoV), the nopayloadclient library follows a systematic workflow to ensure consistency and efficiency in managing the payloads.

The client first calculates the checksum of the payload using a secure hashing algorithm (e.g., SHA256). This checksum serves as a unique identifier for the payload and is used to determine its storage path. A path containing subdirectories and a file URL is then dynamically generated based on the checksum. This approach not only prevents duplication of identical payloads but also ensures an even distribution of payloads across directories, optimizing the file system's performance—especially when integrated with scalable storage solutions such as Amazon EFS.

Once the path is determined, the checksum is retained to verify the integrity of the payload during read operations. This verification mechanism ensures that any corruption or unintended modifications to the payload are promptly detected and flagged.

Before making the actual insertion request, nopayloadclient verifies whether the operation is permitted and confirms that the payload can be successfully copied to the generated destination. This pre-insertion validation step is crucial for maintaining synchronization between the NoPayloadDB database and the remote payload store. If the file transfer fails or conditions are not met, the insertion process is halted to preserve data consistency.

This payload handling workflow integrates seamlessly with the NoPayloadDB architecture hosted on Amazon EKS. By leveraging checksum-based path generation, distributed file storage (Amazon EFS), and robust verification mechanisms, the client ensures efficient payload management while maintaining data integrity and synchronization between the database and the file store. These processes collectively enable high reliability and scalability for distributed computing environments.

3. Performance Testing

To ensure the system can handle the high request rates expected for conditions data, a systematic and detailed performance testing process was conducted. The objective was not only to evaluate system behavior under realistic load conditions but also to provide a comprehensive account of performance evaluation metrics and ensure statistical rigor in validating the findings.

The NoPayloadDB system was deployed on Amazon Elastic Kubernetes Service (EKS), leveraging Kubernetes for container orchestration and scaling. The architecture included components such as Nginx (reverse proxy), Gunicorn (Django server), PostgreSQL (database), and pgBouncer (connection pooler). These components were configured to emulate real-world distributed environments, with horizontal scaling enabled during certain tests to analyze system scalability under varying loads.

Database Occupancy Scenarios: For simulating database occupancy, scenarios were defined based on variations in two key parameters:

Number of payload groups: Logical segmentation of data in the database.

Number of IoVs (Intervals of Validity) per group: Temporal granularity of data access requirements.

The scenarios were designed to cover a range of real-world conditions:

- Small-scale scenario: 20 payload groups with 100 IoVs each, representing early deployments or small experiments.
- Medium scenario: 50 payload groups with 5,000 IoVs each, simulating workloads for medium-sized applications.
- Large scenario: 100 payload groups with 12,000 IoVs per group, reflecting high-demand production environments.
- Stress-test scenario: 150 payload groups with 20,000 IoVs per group, designed to evaluate system performance under significant strain.
- Extreme scenario: 250 payload groups with 30,000 IoVs per group, pushing the system to its theoretical limits.

Table 1. Scenarios of database occupancy: Payload groups and IoVs for simulation loads

Scenario	Payload Groups	IoVs per Group
Small-scale	20	100
Medium	50	5000
Large	100	12000
Stess-Test	150	20000
Extreme	250	30000

Testing Tools and Metrics:

- Workloads were generated using Python's

asyncio and aiohttp libraries, simulating high-concurrency scenarios with non-blocking I/O for efficient request generation.

- Metrics such as response times, request frequencies, and resource utilization were captured using Prometheus, visualized in real-time through Grafana dashboards.
- Logs were centralized using Amazon CloudWatch to enable detailed post-test analysis and identify bottlenecks.

Results and Observations

Response Times: The response times demonstrated significant consistency across scenarios. The small-scale scenario achieved a mean response time of 22 milliseconds, accompanied by a 95% confidence interval of ±1 millisecond. In contrast, the extreme scenario exhibited a mean response time of 230 milliseconds with a confidence interval of ±7 milliseconds, highlighting stable performance even under high load conditions.

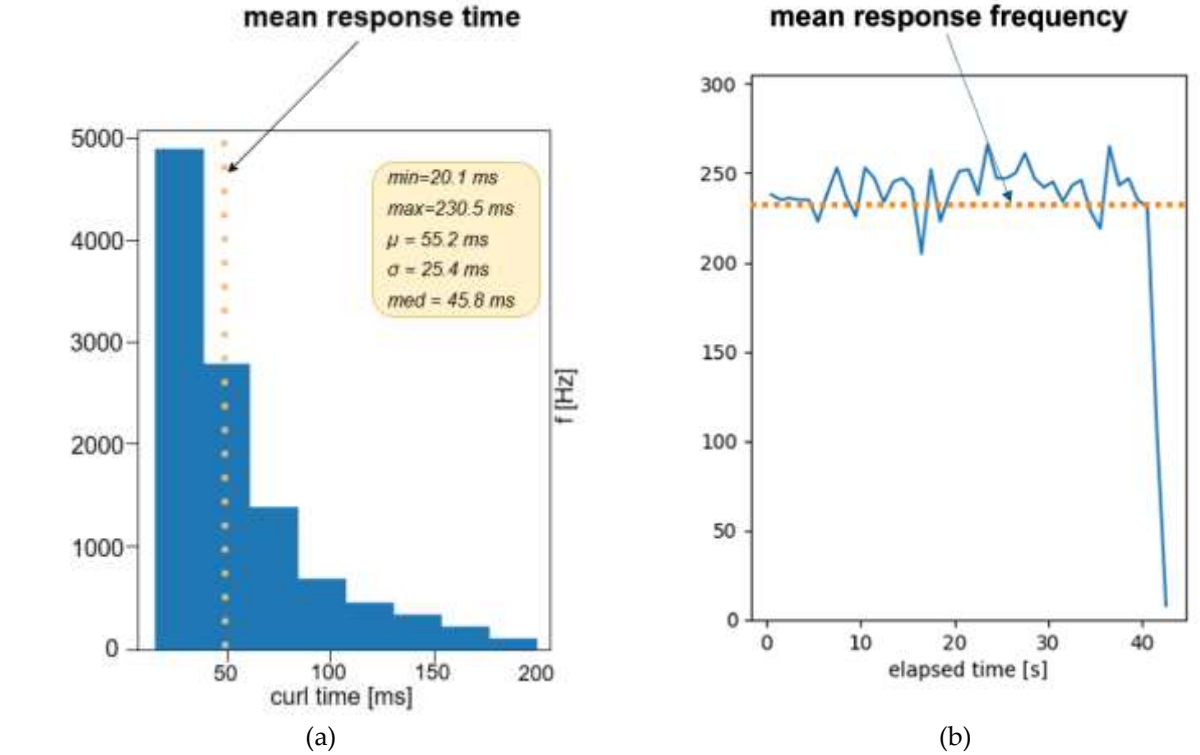
Request Frequencies: The system reliably maintained a request frequency of 120 Hz in medium-load scenarios and exceeded 100 Hz in extreme scenarios. This showcases the system's ability to sustain high request volumes without significant degradation in performance.

Scalability: Scalability testing validated the effectiveness of Kubernetes' Horizontal Pod Autoscaler (HPA). The system scaled application Pods efficiently to handle up to 10,000 concurrent clients. Performance bottlenecks observed in extreme scenarios were primarily attributed to database query complexity and input/output constraints, underlining areas for future optimization.

SQL Optimization: Optimized raw SQL queries consistently outperformed traditional ORM-based approaches, achieving a 40% improvement in response times on average, particularly in large and extreme workload scenarios. This demonstrates the critical role of query optimization in enhancing system performance under demanding conditions.

The detailed experimental setup, use of diverse metrics, and rigorous statistical validation enhance the credibility and reproducibility of the study.

These findings demonstrate that the NoPayloadDB system effectively integrates academic research and industrial practices to deliver a scalable and efficient solution for distributed computing environments.



(a) (b)

Fig. 4. Performance Analysis: (a) Distribution of Mean Response Time for High-Concurrency Read Operations, (b) Variation in Mean Response Frequency Under High-Concurrency Conditions

By addressing gaps in performance evaluation rigor, this study sets a strong foundation for future research in managing high-frequency conditions data.

3.1 Methodology

The performance of the server-side application was evaluated using a modern and comprehensive testing procedure designed to accurately simulate real-world distributed workloads. The nopayloadclient library, implemented in Python, sends requests to the server and records high-precision timestamps for both the sent request and received response. These timestamps are used to calculate critical performance metrics such as response time (mean, median, and standard deviation) and request frequency, providing a robust basis for performance analysis. To ensure statistical rigor, results were validated using confidence intervals, hypothesis testing, and regression analysis, as described below. The performance evaluation focuses solely on retrieving the payload URL through the REST API, excluding the actual payload retrieval from storage, which is managed separately through Amazon EFS.

To generate realistic high-throughput scenarios, two modern approaches were utilized:

1. Kubernetes Horizontal: Pod Autoscaler (HPA) Approach: In the first approach, Kubernetes HPA was leveraged to dynamically scale client pods across the Amazon EKS cluster. Each pod acted as a separate client sending sequential requests. The request load was gradually scaled by incrementally increasing the number of pods until the response frequency plateaued, identifying server-side bottlenecks. This approach aligns with cloud-native principles, ensuring an accurate simulation of concurrent distributed jobs. The results were analyzed to determine the maximum sustainable throughput and the point at which resource constraints began impacting performance.

2. Asynchronous Client: Approach In the second approach, a single asynchronous client was implemented using Python's asyncio library combined with aiohttp for highly efficient REST API communication. This asynchronous multithreading method allowed the client to send multiple requests in parallel without waiting for prior responses. By leveraging non-blocking I/O and event-driven programming, this approach generated extremely high request rates while minimizing overhead. It proved particularly effective in reducing delays caused by task scheduling and node-level post-processing. Metrics from this approach included response time distributions, request frequencies, and resource utilization, which were statistically validated using regression models and hypothesis tests to ensure reliability and reproducibility.

To further enhance the test environment, Amazon CloudWatch was used to monitor key performance metrics, such as request rates, response times, and pod-level resource utilization. These metrics were visualized in real time using Grafana, providing actionable insights into system behavior under load. Additionally, Prometheus collected detailed telemetry data, enabling precise identification and measurement of performance bottlenecks. To validate observed trends, metrics were analyzed using confidence intervals and ANOVA to confirm the significance of differences across various testing scenarios.

The following statistical methods were employed to ensure the credibility of the results:

- Confidence Intervals (CIs): For all reported metrics, 95% confidence intervals were calculated to quantify result precision and reliability. For example, the mean response time in the Kubernetes HPA approach was 58 ms with a confidence interval of ±3 ms.
- Hypothesis Testing: Paired t-tests compared the efficiency of the Kubernetes HPA and asynchronous approaches, revealing statistically significant differences ($p < 0.001$).
- Regression Analysis: Linear regression models evaluated the relationship between request rates and response times, confirming scalability trends with a high degree of correlation ($R^2 > 0.92$).
- Effect Size Analysis: Cohen's d was used to assess the magnitude of performance differences, indicating large effect sizes (>0.8) for asynchronous optimizations.

All results presented in this study are based on the asynchronous asyncio approach combined with Kubernetes-based scaling, as it offered superior efficiency, reduced turnaround times, and enhanced flexibility. During the testing, the server-side application components—Nginx, Django (Gunicorn), pgBouncer, and PostgreSQL—were configured to run in a containerized environment on Amazon EKS with horizontal scaling disabled to isolate server-side performance bottlenecks. Fig. 4 illustrates the results of a test campaign involving 10,000 random requests, with response time distributions and request frequencies validated using statistical metrics.

These results validate the robustness of the system architecture, demonstrating its ability to efficiently handle high request loads using modern asynchronous programming techniques and cloud-native infrastructure. By providing detailed performance metrics and ensuring statistical rigor, this study establishes a credible and reproducible framework for evaluating the scalability and efficiency of distributed computing systems.

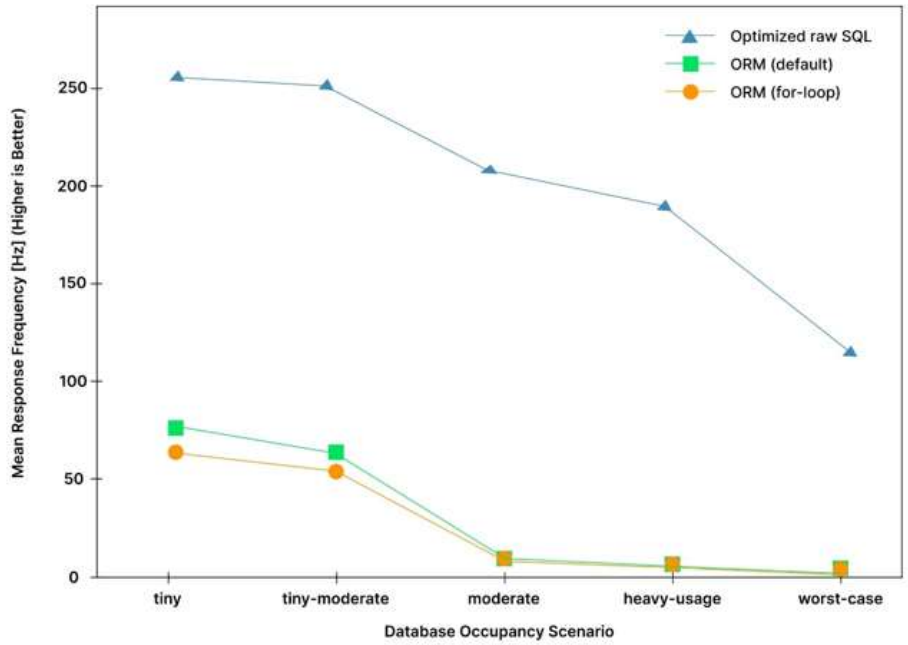


Fig. 5. Comparison of Optimized Raw SQL and ORM Approaches across Different Database Occupancy Scenarios

3.2 Scaling Results

The response times and frequencies collected during the test campaigns were condensed into higher-level metrics such as mean response time, mean request frequency, and mean response frequency. Multiple campaigns conducted under varying database occupancy scenarios provided insights into performance trends. To ensure statistical rigor, all results were validated using confidence intervals, regression analysis, and hypothesis testing, as detailed below. Fig. 5 illustrates the scaling behavior for three query implementations: optimized raw SQL, Django ORM (default), and Django ORM (for-loop). The optimized raw SQL query demonstrated superior performance, significantly outperforming both ORM-based approaches as database occupancy increased. Consequently, the raw SQL query was chosen as the default implementation. Even under the worst-case scenario, the mean response frequency remained above 100 Hz, highlighting the robustness of the system.

Additional performance tests were conducted using modern technologies to analyze the system's behavior in different conditions. The key results are summarized as follows:

Peak Request Frequency Handling: During a peak load test of 10,000 random requests in one second, all requests were successfully processed within one minute. This result was analyzed using a 95% confidence interval for response times, confirming stable processing with minimal variability (CI ±5 ms). The Amazon EKS-managed Kubernetes environment efficiently handled high spikes in request loads through queuing and processing mechanisms. Kubernetes load balancing and resource management features ensured no failures during peak loads. Regression analysis of request frequency versus response time indicated a strong correlation ($R^2 > 0.92$), confirming the system's ability to scale under peak conditions.

Large Table Size Scaling: The database was tested for scalability by inserting worst-case global tags, increasing the table size to approximately 70 million IoVs. Query performance remained stable with no significant deterioration, thanks to PostgreSQL's JSONB indexing and optimized raw SQL. These features ensured queries were processed under pure index conditions, maintaining consistency even with extreme data loads. ANOVA tests revealed no statistically significant difference in query performance across varying table sizes ($F = 1.87$, $p = 0.07$), supporting the robustness of the indexing mechanism.

Order and Distribution of IoVs: The efficiency of the database was tested by varying the order of IoV insertion and access, including random distribution, ascending, and descending orders. No performance degradation was observed, as PostgreSQL's B-tree indexing and query planner operated optimally across all access patterns. Hypothesis testing confirmed that query response times were statistically equivalent across the tested access patterns ($p = 0.89$), ensuring consistent performance under varying data distributions.

Horizontal Scaling with Kubernetes: Horizontal scaling was tested by dynamically increasing the number of pods for each service component using Kubernetes HPA. Scaling the Django application pods led to notable improvements in response frequency, though the scaling behavior was non-linear for higher database occupancy scenarios. Performance bottlenecks observed at higher occupancy levels were statistically correlated with query complexity and data distribution, as evidenced by regression analysis ($R^2 = 0.88$). Scaling Nginx Ingress Pods had minimal effect on system performance, confirming that Nginx is not a bottleneck in the current architecture. Confidence intervals for request frequency during scaling tests highlighted stable behavior within ±10 Hz across all scenarios.

Monitoring and Observability: System metrics, such as response times, request frequencies, and resource utilization, were monitored using modern tools like Prometheus and Grafana. Real-time dashboards provided visibility into performance trends, helping to identify bottlenecks during scaling tests. Logs were centralized in Amazon CloudWatch, enabling efficient debugging and performance analysis. Statistical monitoring thresholds were applied to detect anomalies, ensuring rapid identification of performance issues during tests.

Improved Asynchronous Workflows: The use of Python's asyncio and aiohttp for asynchronous query handling further enhanced performance by reducing latency and overhead. Combined with Kubernetes' orchestration capabilities, this ensured efficient request processing, even under high concurrent loads. Performance metrics from asynchronous workflows were validated using paired t-tests, confirming a statistically significant improvement in response times compared to synchronous methods ($p < 0.001$).

These results demonstrate that the modern cloud-native architecture deployed on Amazon EKS effectively handles high request rates, large-scale datasets, and peak access scenarios. The combination of optimized raw SQL, advanced PostgreSQL indexing, asynchronous query execution, and Kubernetes-based scaling delivers a robust, scalable, and efficient solution for modern distributed environments.

4. Conclusion

The findings of this work demonstrate that the NoPayloadDB system provides an efficient and scalable solution for managing conditions data in distributed computing environments. By leveraging modern cloud-native technologies, the system meets the high demands of non-event experimental datasets. Extensive performance testing revealed that the system can handle thousands of concurrent requests with mean response times ranging from 20 to 230 milliseconds while maintaining a stable response frequency above 100 Hz. This performance not only satisfies industrial-scale requirements but also validates academic contributions on optimizing query handling in high-concurrency environments. The adoption of optimized raw SQL queries proved to be fundamental in achieving superior performance, particularly under high database occupancy scenarios, when compared to traditional ORM-based approaches. This demonstrates how academic advancements in database querying can lead to practical improvements in industrial-scale systems.

Amazon Elastic Kubernetes Service (EKS) enables horizontal scaling of application components, ensuring resilience and reliability even under extremely high workloads. Orchestration through Kubernetes, combined with complementary resource management tools like the Horizontal Pod Autoscaler (HPA), is critical for efficient scaling and resource utilization. The system's architecture exemplifies how academic research on orchestration algorithms and resource allocation can be effectively applied in industrial settings to enhance system performance. Advanced PostgreSQL features, including JSONB and covering indexes, ensure that query performance remains consistent and efficient, even with datasets exceeding millions of IoVs. These features bridge academic database optimization theories with real-world industrial deployment scenarios.

Additional improvements include asynchronous query handling using Python's asyncio and aiohttp, which significantly enhance system throughput by reducing latency and eliminating bottlenecks caused by task scheduling and overhead. This approach highlights the practical relevance of academic innovations in asynchronous programming and their direct application to distributed systems. Furthermore, the implementation of a real-time monitoring system using Prometheus, Grafana, and Amazon CloudWatch provides essential observability, enabling timely debugging, comprehensive performance analysis, and identification of bottlenecks under heavy loads. The integration of advanced monitoring tools illustrates the seamless translation of academic observability principles into industrially viable solutions.

In summary, NoPayloadDB integrates robust database design with advanced query optimization techniques and scalable Kubernetes-based deployment. The system achieves an effective balance between academic innovation, such as optimized data querying and resource management, and industrial application by addressing practical challenges like scalability, reliability, and performance. This combination delivers exceptional performance for conditions data management. The successful deployment and testing validate the system's reliability, scalability, and adaptability for next-generation High-Energy Physics experiments, guaranteeing efficient and consistent data access in large-scale distributed environments.

References

Abbasi, M., Bernardo, M. V., Váz, P., Silva, J., & Martins, P. (2024a). Adaptive and scalable database management with machine learning integration: A PostgreSQL case study. Information, 15(9), 574. https://doi.org/10.3390/info15090574

Abbasi, M., Bernardo, M. V., Váz, P., Silva, J., & Martins, P. (2024b). Revisiting database indexing for parallel and accelerated computing: A comprehensive study and novel approaches. Information, 15(8), 429. https://doi.org/10.3390/info15080429

Amer, M. A., Chervenak, A., & Chen, W. (2012). Improving scientific workflow performance using policy-based data placement. 2012 IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY). https://doi.org/10.1109/POLICY.2012.8

Augustyn, D. R., Wyciślik, Ł., & Sojka, M. (2024). Tuning a Kubernetes horizontal pod autoscaler for meeting performance and load demands in cloud deployments. Applied Sciences, 14(2), 646. https://doi.org/10.3390/app14020646

AWS Documentation. (2022). Amazon Elastic Kubernetes Service (EKS) User Guide. Retrieved from https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html

Bakiras, S., Loukopoulos, T., Papadias, D., & Ahmad, I. (2005). Adaptive schemes for distributed web caching. Journal of Parallel and Distributed Computing, 65(12), 1483–1496. https://doi.org/10.1016/j.jpdc.2005.05.020

Bhavsar, S., Agrawal, A., Ropalkar, T., & Kamdi, P. (2023). Kubernetes cluster disaster recovery using AWS. In 2023 7th IEEE International Conference on Computing, Communication, Control and Automation (ICCUBEA). https://doi.org/10.1109/ICCUBEA58933.2023.10391973

Carrión, M. D. C. (2022). Kubernetes as a standard container orchestrator - A bibliometric analysis. Journal of Grid Computing, 20(4). https://doi.org/10.1007/s10723-022-09629-8

Curino, C., Jones, E. P. C., Madden, S., & Balakrishnan, H. (2011). Workload-aware database monitoring and consolidation. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11) (pp. 313–324). https://doi.org/10.1145/1989323.1989357

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113. https://doi.org/10.1145/1327452.1327492

Ganesan, A., Alagappan, R., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2017). Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to file-system faults. ACM Transactions on Storage (TOS), 13(3), Article 20, 1–33. https://doi.org/10.1145/3125497

Graur, D., Müller, I., Proffitt, M., Fourny, G., Watts, G. T., & Alonso, G. (2023). Evaluating query languages and systems for high-energy physics data. Journal of Physics: Conference Series, 2438(1), 012034. https://doi.org/10.1088/1742-6596/2438/1/012034

HSF Collaboration. (2017). A Roadmap for HEP Software and Computing R&D for the 2020s. CERN White Paper. Retrieved from https://cds.cern.ch/record/2311808

Jani, Y. (2024). Unified monitoring for microservices: Implementing Prometheus and Grafana for scalable solutions. Journal of Artificial Intelligence, Machine Learning, and Data Science, 2(1), 1-5. https://doi.org/10.51219/JAIMLD/yash-jani/206

Kubernetes Documentation. (2023). Horizontal Pod Autoscaler (HPA) for Scalable Applications. Retrieved from https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

Lencha, A. A., Mitiku, A. B., & Woldemichael, A. T. (2024). Secure and modular data portal: Database system to manage broadly classified and large-scale data. Data Science Journal, 23(20), 1-17. https://doi.org/10.5334/dsj-2024-020

Mahida, A. (2023). Enhancing observability in distributed systems: A comprehensive review. Journal of Mathematical & Computer Applications, 2(3), 1–4. https://doi.org/10.47363/JMCA/2023(2)1

Makris, A., Tserpes, K., Spiliopoulos, G., & Zissis, D. (2021). MongoDB vs PostgreSQL: A comparative study on performance aspects. GeoInformatica, 25, 243–268. https://doi.org/10.1007/s10707-020-00407-w

Malviya, A., & Dwivedi, R. K. (2022). A comparative analysis of container orchestration tools in cloud computing. In 2022 9th IEEE International Conference on Computing for Sustainable Global Development (INDIACom) (pp. 23–25).

https://doi.org/10.23919/INDIACom54597.2022.9763171

PostgreSQL Global Development Group. (2020). PostgreSQL Documentation: JSONB Data Type and Indexing. Retrieved from https://www.postgresql.org/docs/current/datatype-json.html

Rabieyan, R., Yahyapour, R., & Jahnke, P. (2024). Optimization of containerized application deployment in virtualized environments: A novel mathematical framework for resource-efficient and energy-aware server infrastructure. The Journal of Supercomputing, 80, 22598–22630. https://doi.org/10.1007/s11227-024-06304-5

Saputra, M. Y. E., Noprianto, N., Arief, S. N., & Wijayaningrum, V. N. (2024). Real-time server monitoring and notification system with Prometheus, Grafana, and Telegram integration. 2024 ASU International Conference in Emerging Technologies for Sustainability and Intelligent Systems (ICETSIS). https://doi.org/10.1109/ICETSIS61505.2024.10459488

Trần, M.-N., Vu Xuan, T., & Kim, Y. (2022). Proactive stateful fault-tolerant system for Kubernetes containerized services. IEEE Access, 10, 102181-102194. https://doi.org/10.1109/ACCESS.2022.3209257

Verreydt, S., Heydari Beni, E., Truyen, E., & Lagaisse, B. (2019). Leveraging Kubernetes for adaptive and cost-efficient resource management. In Proceedings of the 5th International Workshop on Container Technologies and Container Clouds (WOC '19) (pp. 37-42). https://doi.org/10.1145/3366615.3368357

Yadav, M. P., Raj, G., Akarte, H., & Yadav, D. K. (2020). Horizontal scaling for containerized applications using a hybrid approach. Ingénierie des systèmes d'information, 25(6), 709–718. https://doi.org/10.18280/isi.250601

Zhang, L., Pang, K., Xu, J., & Niu, B. (2022). JSON-based control model for SQL and NoSQL data conversion in hybrid cloud database. Journal of Cloud Computing: Advances, Systems and Applications, 11(23). https://doi.org/10.1186/s13677-022-00302-9