Electronic Research Archive

ERA, 34(12): 9462–9484.
DOI: 10.3934/era.2026397
Received: 09 April 2026
Revised: 26 July 2026
Accepted: 07 September 2026
Published: 18 November 2026

https://www.aimspress.com/journal/era

Research article

Hybrid database for cloud microservice architecture to enhance real-time financial technology systems' performance

Hisham AbouGrad* and Emmanuel Iyahen

Department of Computer Science and Digital Technologies, School of Architecture Computing and Engineering, University of East London, Docklands, London, UK

* **Correspondence:** Email: H.AbouGrad@uel.ac.uk.

Abstract: Financial microservices database systems need the transactional guarantees of the relational database, but at the same time, they should ensure the low latency of real-time services. A typical way to resolve such issues is developing a hybrid database model that integrates an in-memory database to compliment the relational database. In many cases, the level of performance increase is neither precisely estimated, nor is the mechanism of this increase explained. This research article examined both issues for a transaction-based microservice running on Amazon elastic container service by comparing the performance between the PostgreSQL-only database setup and the hybrid PostgreSQL + Redis database setup. Both setups were tested under closed user load of 5, 20, and 50 concurrent users generated with Locust. A unified performance model with normalized throughput, mean response time, and failure rate is introduced. The hybrid setup achieved high-performance at all load levels including the highest load of 50 concurrent users. At 50 concurrent users, the throughput grown from 20.37 RPS to 23.40 RPS, and mean response time drops from 469.99 ms to 158.48 ms, while the failure rate is reduced sixfold, from 0.13% to 0.02%. Although the percentile analysis resulted in recurring tail latency jumps in both PostgreSQL-only and hybrid setups, these were less pronounced in the case of the hybrid setup. The analysis of failures proves that PostgreSQL-only failures are caused by connection pool overflow. Using Little's Law, the mean connection demand was calculated to be 9.6 connections for PostgreSQL-only against 3.7 connections for the hybrid, and 48% against 19% for the pool maximum. The research findings prove that PostgreSQL-only mean connection time decreased through the integrated cache in the hybrid setup, which increases financial microservices database systems performance and overall FinTech solutions capability.

Keywords: database design; hybrid database architecture; AWS elastic container microservices; database system performance optimization; real-time cloud database

1. Introduction

Financial technology (FinTech) solutions have been used increasingly in a wide range of financial activities throughout the world for businesses, including customer-to-customer, business-to-customer, and business-to-business models. Conversely, due to the growth in the adoption of FinTech platforms, they are expected to be secure and reliable, scale up efficiently, and respond to queries quickly [1]. This is not easy for monolithic architecture because using a single database limits its ability to scale and creates a bottleneck for its performance. This problem becomes even more complex when it comes to financial operations that need both integrity and fast performance. Indeed, microservices cloud computing technology is able to address such technical issues through the separation of FinTech architecture into small self-contained services, which can be deployed and scaled separately [2]. However, it is quite important to note that performance increases that can be achieved due to such an approach heavily depend on the database management system (DBMS) used for services operation.

Further, relational database management systems (RDBMS), such as PostgreSQL, provide strong atomicity, consistency, isolation, and durability (ACID) properties (features) that ensure their position as the safest option for keeping track of financial transactions [3]. As load increases, however, latency increases while throughput remains the same [4]. On the other hand, NoSQL databases, such as Redis, provide fast response time and scalability but generally do not provide the transactional guarantees necessary for finance processing [5]. While each database type alone cannot meet the criteria and taking into account the large number of financial transactions that occur in FinTech systems, there is an emerging need for combining RDBMS and NoSQL DBMS in a microservices architecture. In such an approach, the RDBMS is able to ensure transactions, while the NoSQL DBMS is able to manage the latency-dependent reads. Furthermore, the utilization of a hybrid database (HDB) architecture is able, in theory, to serve thousands of users concurrently, using the reliability of the relational database system and the more speed of the non-relational database system. This type of architecture applies a hybrid cloud database model through the construction of microservices of accounts and transactions utilizing PostgreSQL and Redis [6]. The research study applied this research framework by measuring and diagnosing the degree to which the hybrid database FinTech solution improves DBMSs' reliability, scalability, and performance.

Overall, the integration of HDB technologies within the microservices provides a compelling solution to address the challenges of transactional consistency and scalability in modern FinTech business environments. As HDB models provide the stability and reliability required in financial processes and procedures; they enable the DBMS to ensure its efficiency even under increasing loads, which is used to utilize the benefits of both types of database technologies. Thus, evaluating how these HDB models could be effectively designed and implemented would contribute to the growing pool of literature and deliver ideas to guide the development of future FinTech DBMS solutions.

2. Literature review and related work

The study focused on reviewing previous literature on data management techniques applied in

FinTech systems and business environments, with an emphasis on microservices' architecture. The study explores database systems and their application to satisfy FinTech's increasing requirements in terms of performance, scalability, and data consistency. The literature review is organized into key areas associated with FinTech microservices and DBMSs in the following sequence: Data management in microservices, relational databases (SQL), and nonrelational (NoSQL) databases in FinTech. This includes identifying these key areas' strengths and weaknesses. Overall, this section provides background for introducing the proposed HDB model, which supports identifying the current research gaps and provides a summary of the key related methods.

2.1. Data management in microservices

Research studies about data management in microservices' architecture are still evolving, particularly in sectors with high requirements, such as FinTech systems [3,4]. The decentralized nature of microservices introduces significant complexity in data storage and consistency. The adoption of heterogeneous databases in microservices is on the rise. For example, the study of Pantelic et al. [7] showed how polyglot persistence is prevalent in microservices, where multiple database systems are implemented by different services, depending on the workload requirements.

Relational databases and non-relational databases have their individual advantages with respect to scalability, queries, and resource utilization, which makes the selection of the database type an important consideration for data management [5,8]. This approach, which promotes heterogeneity and optimization, was highlighted in recent studies as being widely adopted in FinTech and overall information technology (IT) industries. Similarly, research on polyglot persistence proves that the integration of relational, key value, and document stores improves the performance and flexibility of applications but may also complicate operations and produce consistency problems.

2.2. Relational database systems

Traditional relational DBMSs, such as PostgreSQL, remain indispensable in FinTech systems because they ensure the strict ACID properties of the data [5, 9]. These properties ensure the correctness and reliability of data, which is strongly required in financial domains. These guarantees are essential in financial environments where even minor anomalies can lead to significant monetary errors or compliance failures. Conversely, the limitations of traditional relational database systems have been demonstrated by the increased size and complexity of the workload in finance applications. The reliability of relational databases in terms of data integrity is unmatched; however, they may be limited in terms of performance when faced with high concurrency and transaction volumes, as indicated by several studies, including Ionescu et al. [10]. These database design scenarios are characterized by low latency and high throughput requirements.

Despite all their strengths, the inherent scalability problems of relational database systems are beginning to emerge in the current FinTech processing models. According to recent studies, standard relational designs face significant concurrency problems due to the massive volume of transactions, particularly in high-frequency environments and real-time financial analysis [11]. Moreover, other current studies have found that the vertical scaling approaches associated with RDBMSs cannot meet the requirements of modern FinTech applications. On their own, RDBMSs cannot cope with the exponential growth of transaction volumes and the diversity of data, which makes such database

systems less efficient. This necessitates the exploration of additional data management strategies that can go beyond the capabilities of traditional relational database systems.

2.3. Non-relational cloud database systems

Several research studies have been conducted to investigate the ability of nonrelational (NoSQL) databases, such as MongoDB, Cassandra, and Redis, to solve scalability challenges in relation to relational databases. The success achieved by NoSQL databases in solving these challenges is attributed to efficient data access mechanisms, the scalability of the schemas, and the ability to scale horizontally [12]. Experimental studies by Jawale et al. [13] and Topcu et al. [14] indicated that NoSQL databases perform better than relational databases in terms of read/write intensive operations, which are common to FinTech applications, such as transaction streaming, caching, and analytics.

With respect to FinTech systems, proof of the efficiency of NoSQL databases was provided in recent studies by AbouGrad et al. [15] and Bayya [16], which, respectively, addressed implementing a real-time NoSQL database (MongoDB) for cloud-based contactless mobile payment systems, data-driven predictive analytics, and decision-making in FinTech and high-throughput data pipelines. These studies revealed that real-time databases based on NoSQL are capable of supporting efficient, rapid, and scalable processes, especially in high-workload environments characterized by requirements for prompt data access and low latency. Conversely, availability and partition tolerance are prioritized over strict consistency. Since consistency is a critical necessity for financial systems, NoSQL database systems are not used independently in many financial domains.

Lastly, regarding HDB implementations in finance and its related domains, apart from the polyglot persistence literature discussed earlier, there have been several recent industry-focused research studies that have explored the implementation of hybrid SQL+NoSQL systems within the finance domain. For instance, Bagam [17] proved the strategic use of a hybrid data architecture that integrates both relational and nonrelational database systems within several case studies on financial services companies and identified improvements in transaction processing capabilities and query response times due to specially designed data-tiering between the two models of data storage architecture. This led to a conclusion that is similar in its overall orientation to the findings of the current research, although derived through an industry case analysis rather than performance modeling and testing. Moreover, Ionescu and Diaconita [18] reviewed how cloud computing and advanced data management systems, along with hybrid data management methods, are transforming the decision-making infrastructure into finance and contextualized the hybrid data architecture within a broader paradigm of artificial intelligence (AI) driven financial database platforms. Compared with these studies, this research study contributes a performance modeling and load-testing-based evaluation of a particular SQL+NoSQL combination using PostgreSQL and Redis database management systems.

2.4. The research gap

The tradeoff that exists between transactional integrity and scalability within database architecture design has been illustrated and critically analyzed. It is obvious that NoSQL database systems can deliver high levels of performance and scalability, but at the expense of strict data consistency, whereas relational databases deliver strong consistency but lack scalability. This has led to the emergence of HDB designs, which seek to exploit the complementary strengths of both models,

owing to their trade-offs. However, the number of empirical analyses carried out regarding the effectiveness of such database systems for FinTech transactions is limited. Indeed, this study has focused on filling this research gap with a scientific and practical evaluation of an HDB design that utilizes both PostgreSQL (an RDBMS) and Redis (a NoSQL cloud computing database). The HDB model was tested on real FinTech transaction cases within a microservice cloud computing environment. The study makes a meaningful contribution to the field by proving polyglot persistence and designing scalable financial data platforms based on various key performance metrics, such as throughput, latency, and failure rate, as required by modern FinTech systems.

3. Methodology and methods

The research applied design science research (DSR) methodology to guide the development and evaluation of the proposed HDB design. This approach mainly focuses on developing new software and IT artifacts and empirically validating them [19]. The main artifact being developed in this research study is an HDB model designed for FinTech microservices. The system architecture consists of two microservices, namely; the account service, which is concerned with managing user accounts, and the transaction service, the primary focus of this research is investigating the performance of money (fund) transactions.

Further, two database setups were compared in order to determine their influence on the transaction service. For the first database setup, referred to as RDBMS-only, the transaction service retrieves user account data from the RDBMS before making any transfers between accounts. For the second database setup, known as hybrid, which introduces Redis as a secondary cache layer, user account data are stored in the Redis in-memory cache, and the transaction service retrieves the data from the Redis cache, not from the RDBMS, in order to minimize the number of database queries. Both database setups were tested under varying load using Locust, an open-source concurrency testing tool for Python written HDB model and FinTech software system [20].

Furthermore, there were three key performance metrics measured. Throughput (θ) is measured as requests per second (RPS) and refers to the number of transactions being processed in a certain period of time. Average response time (T) refers to the average amount of time taken to process each transaction. The failure rate (λ) refers to the percentage of transaction requests that fail under load. In all, these three measures allow for a balanced measure of performance, which will facilitate the ability to compare the results of the RDBMS and HDB system. Previous study in microservices architecture and database performance optimization have verified the same method for optimizing service responsiveness and scalability [21].

3.1. Mathematical formulation of performance evaluation

FinTech business transactions require higher throughput, lower latency, and lower failure rates for better performance. To achieve a formal and quantitative comparison of the RDBMS-only and HDB configurations, the database system performance is modeled as a weighted function of the three performance metrics as follows:

$$P(Si) = w1 \bullet \vartheta(Si) - w2 \bullet T(Si) - w3 \bullet \lambda(Si), \tag{3.1}$$

where $Si \in \{S1, S2\}$ represents the system's configuration, i.e., S1 is the RDBMS-only configuration

and S2 is the RDBMS + Redis configuration; $\theta(Si)$ denotes throughput; $T(Si)$ denotes the average response time; and $\lambda(Si)$ denotes the failure rate. The coefficients *w1, w2, w3* > 0 are the weight (*w*) factors assigned to each performance metric based on their relative importance. This ensures that throughput contributes positively to the system's performance, while latency and failure rate impose penalties on the system.

The weights *w1*, *w2*, and *w3* are all assigned a value of 1, indicating that each of the weights has equal priority for evaluating the throughput, response time, and failure. Indeed, applying such weights directly to the values obtained for the metrics (i.e., θ in RPS, *T* in ms, λ in %) without further modification would be a dimensionally inappropriate approach to combine metrics that vary significantly in their order of magnitude; specifically, because *T* (in hundreds of ms) would numerically dominate θ (in single to double digits of RPS) even if all the weights had the same value of 1. For this reason, the following procedure is used to normalize the metrics to their percentage of change from the RDBMS-only configuration (S1):

$$\theta_{norm} = \big(\theta(S2) - \theta(S1)\big)/\theta(S1), \tag{3.2}$$

$$T_{norm} = \big(T(S1) - T(S2)\big)/T(S1), \tag{3.3}$$

$$\lambda_{norm} = \big(\lambda(S1) - \lambda(S2)\big)/\, max(\lambda(S1), \varepsilon), \tag{3.4}$$

with a small constant ε introduced into the denominator of λ_{norm} to prevent a division by zero error in tests in which λ(S1) = 0, which is the case for Tests 1 and 2 (see Table 1). The normalized value of each component is a dimensionless relative improvement, and so *P(Si)* in Eq (3.1) may then be calculated using θ_{norm}, *Tnorm*, and λ_{norm} instead of the unnormalized values, such that the choice of the weights *w1*, *w2*, and *w3* is equivalent among the metrics and not skewed toward the raw metric having the largest value. The relative performance improvement of the proposed system is defined as

$$\Delta Pk = P(S2) - P(S1), \tag{3.5}$$

This can be expanded to

$$\Delta Pk = w1(\theta 2 - \theta 1) + w2(T1 - T2) + w3(\lambda 1 - \lambda 2), \tag{3.6}$$

where *k* denotes the test cases. The HDB configuration outperforms the RDBMS-only configuration if ΔPk is positive and vice versa. This formula guarantees that 1) θ2 > θ1 contributes positively, since higher throughput is desirable; 2) T2 < T1 contributes positively, since lower latency is desirable; and 3) λ2 < λ1 contributes positively, since a lower failure rate is desirable.

To further quantify the database system's performance and efficiency, a latency-based speedup factor is modeled, given by

$$Speedup = \frac{T(S1)}{T(S2)}, \tag{3.7}$$

This ratio is an indication of how much faster the hybrid system is compared with the RDBMS-only setup. If the ratio is greater than one, there is an improvement in the speed of the system.

Lastly, the equations above provide a more quantitative approach to analyzing the behavior of the database system and meet the goals of efficiency in FinTech complex systems.

Table 1. Results of RDBMS versus RDBMS + Redis.

Test case	Design	θ (req/s)	T (ms)	λ (%)
Test 1	RDBMS (PostgreSQL)	2.34	239.72	0
	RDBMS + NoSQL	2.39	211.22	0
Test 2	RDBMS (PostgreSQL)	9.09	224.00	0
	RDBMS + NoSQL	9.38	212.32	0
Test 3	RDBMS (PostgreSQL)	20.37	469.99	0.13
	RDBMS + NoSQL	23.40	158.48	0.02

3.2. HDB model system architecture

The financial HDB model is made up of three layers; the client layer, which is an interface for client-server communication (Web/Mobile App), the application layer that has the account service and transaction service (Microservices), and the data layer that stores the system data (Persistence and Cache). The overall HDB system's architecture is illustrated in Figure 1.



Figure 1. Overall HDB model's system architecture.

The relational database system comprises two microservices handling financial processes and operations. Account service is responsible for account creation, authentication and account details, while the transaction service executes the fund transfers between accounts. Both services use the PostgreSQL relational database as their primary database as illustrated in Figure 2.

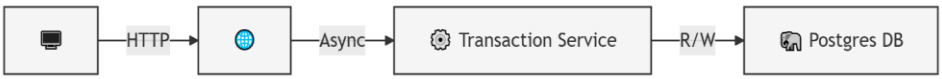


Figure 2. RDBMS-only database setup and configurations.

The HDB configuration integrates Redis to cache account data of users in memory (Figure 3). When a transfer is processed, the transaction service retrieves the required account data from Redis rather than querying PostgreSQL, thus minimizing the read operations to the primary database.

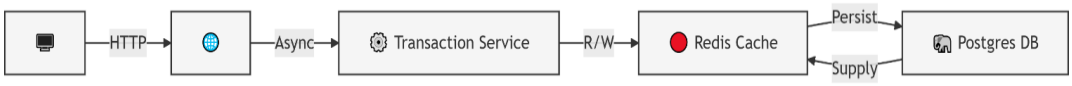


Figure 3. RDBMS with the Redis database setup and configurations.

3.3. HDB design and integration

The PostgreSQL relational database was deployed through Amazon's RDBMS (AWS database), serving as the primary data store where all the structured financial data were stored in a normalized user, account, and transaction tables. This ensures that transactions were always ACID compliant. In the HDB setup, Redis served as the cache for the account table only.

The transaction service utilized in the HDB design setup is different in its architecture from the PostgreSQL-only setup as a result of the secondary cache (Redis) layer added to support PostgreSQL. Both services have been separately implemented and deployed. Upon initialization, the hybrid transaction service requests all of the user account entries from the account service once and cache them as key value pairs in Redis before conducting any load testing; the preloading operation does not take place again in the course of running a test case, and hence Redis starts out fully populated in all the test cases. Lastly, each account record is stored in Redis as a key-value pair, making it possible to retrieve them very fast without adding any read workload on the PostgreSQL database. A transaction data flow in this HDB system setup involves four steps, which include: 1) retrieving account details from Redis; 2) running the transaction algorithm; 3) updating the balances on the PostgreSQL database; and 4) updating the entries on the Redis cache asynchronously to maintain data consistency.

3.4. Experiments, validation, and testing

To ensure isolation, scalability, and realistic cloud execution, the account and transactions services were implemented using the AWS elastic container service (ECS), which runs within a virtual private cloud (VPC) [22,23]. To replicate production environments for FinTech system designs, microservices were containerized to enable controlled and repeatable testing.

The transaction services and account service were containerized as ECS tasks running on AWS with PostgreSQL configured using Amazon's RDBMS and the cache configured using Amazon ElastiCache for Redis; all these components were co-located in the same AWS VPC for all tests conducted during the present study [24]. Co-location was a deliberate methodological decision taken in order to control the impact of adding a caching layer to the transaction service arising from network latency and consistency issues that may arise from a distributed multinode environment.

Three load test scenarios were developed for analysis of the system's performance using the Locust load simulation tool. The first scenario is the baseline, executed with five concurrent users making 300 requests in total to reflect low load on the systems. The second scenario presents a medium-level load test for 20 concurrent users making 2000 requests in total. Test Scenario 3 shows a high load simulated by 50 concurrent users making 5000 requests in total. These test scenarios have been developed with the purpose of identifying the characteristics of the system's behavior under load conditions.

Further, each of the six test cases presented in this research, using PostgreSQL-only and the HDB at all three different load levels, was run on a newly started service instance. There was no carryover of previous processes of the services, such as database connection pools or the Redis cache, between test cases. The PostgreSQL-only service was stopped, and the hybrid service started, with the one-time preloading of the Redis cache from the account service described in Section 3.3, but only after completing and recording the results of the PostgreSQL-only test case for a particular scenario and vice versa for the subsequent scenario. This approach eliminates the possibility that between-

scenario cache buildup or reuse of the connection pool could explain any difference in the results between scenarios, and is further explained in Section 3.5, along with its impact on the results of response time in Section 4.

Furthermore, three key performance metrics are evaluated in this study, which are throughput (θ), average response time (T), and failure rate (λ). The processing capacity of the system is measured using throughput, which indicates the number of transactions processed per second. Response time gives an insight into the system's responsiveness by determining the amount of time taken to process a transaction request. Failure rate indicates the system's reliability when subjected to stress conditions, measuring the number of failed transactions underload. The experiments were all conducted in controlled environments with similar workload features, including the request payload size, zero think time for maximum concurrency, and fixed testing periods, ensuring the reliability and accuracy of the data. To ensure stability in performance and eliminate any startup effects on the system, a warm-up phase was introduced prior to collecting the measurements. The three test cases were carried out for both configurations to establish a direct and fair comparison between the PostgreSQL-only and HDB systems.

Moreover, the process of retrieving account details consists of two operations. The first involves issuing a request to the account service to retrieve a simulated user's account details (mainly the account identifier) at the start of the user's session at the beginning of each test case prior to measuring the window. Hence, the total number of account requests per test case will be equal to the number of users in each test case, which are 5, 20, and 50 for Test Cases 1, 2, and 3, respectively. The second involves account balance retrieval. Since each transfer transaction changes the balance of an account, the current balance value cannot be determined once and used for further transfers but should instead be retrieved anew prior to each transfer in order to make sure that the current state is used and to prevent inconsistent transactions due to the usage of outdated data. Thus, the current balance check is made when processing each transaction creation request inside the timed measurement window based on the account details that were retrieved at the beginning of the user's session. The account balance is retrieved from the Redis cache in the hybrid setup, as described in Section 3.3, but is queried directly from the PostgreSQL database in the PostgreSQL-only (RDBMS) solution. Since the balance check is done as an internal process while processing the transaction transfer request, instead of sending a new request from the client side, it does not count towards the number of requests made in Table 1 [6]. The throughput, response time, and failure rate numbers given in Table 1 are hence solely based on transaction creations (transfer requests), as seen in the individual endpoint breakdown in Table 2. Since the creation of transactions was the only request issued by the Locust load tester during the measurement window, it was configured to be the only task for Locust to execute in this phase.

Table 2. Summary statistics of Test 3 per endpoint (transfer).

Configuration	Requests	Failures	Avg (ms)	Min (ms)	Max (ms)	RPS
RDBMS-only	5000	7 (0.14%)	469.99	140	3162	20.37
Hybrid	5000	1 (0.02%)	158.48	104	1007	23.40

Overall, the experiment design presents a systematic approach to the evaluation of the transaction service performance across different database configurations. This allows for effective evaluation of

the scalability, responsiveness, and reliability features of the database system by progressively increasing load on the system and analyzing key performance criteria.

3.5. Threats to validity and experimental controls

The control measures used in the research study and how they affect the validity and generalizability of its results should be considered. Internal validity with the fresh restart protocol was maintained, where each test case was executed using a newly started service instance without the previous cache, connection pool, or other process information from a prior test case. This ensured that the RDBMS-only transaction service and hybrid transaction services were restarted between each test case execution, for a total of six restarts for all three test loads and two configurations and also ensured that neither any potential buildup of the cache in Redis nor any residual database connections could be responsible for the response time behaviors.

Further, to ensure external validity for single-node co-location and the study scope applied in the experiments by utilizing the account service and transaction service running as AWS ECS tasks, PostgreSQL was set up using Amazon's RDBMS, and the cache was set up using Amazon's ElastiCache for Redis. These were co-located within a single AWS VPC, which was a deliberate choice, and it set the context of the study's comparison. The experiment investigated the impact of adding a secondary caching layer in a Redis database to a single-node PostgreSQL-only (RDBMS) transaction service. It did not test, and was not meant to evaluate, the impact of changing the deployment topology, such as whether the hybrid solution's components should be split across different nodes, or testing the caching solution against another scaling option of the relational database, such as adding a second PostgreSQL node to a read-replica or sharded solution. These are plausible variations of topology that can be further investigated on their own, but neither of them falls within the scope of the research problem, as the study sought to find the impact of a caching layer on a particular PostgreSQL-only benchmark. Hence, the boundaries have been defined explicitly and mention both investigation of the distributed topology and a comparison of a caching approach with horizontal scaling as interesting areas for future research work. The research findings can be interpreted only as an illustration of the impact of the hybrid caching strategy under co-location and single-node topology.

Lastly, to ensure construct validity, the performance model presented in Eq (3.1) utilizes "normalized metric values" rather than raw measurements, as discussed in Section 3.1. This normalization enables the assumption of equal metric weighting by placing throughput, response time, and failure rate on a common scale to allow meaningful and comparable interpretation of their contributions to the overall performance evaluation.

4. Discussion and findings

The study's results and research findings, including a detailed analysis of the results of performance testing obtained from a cloud-native microservice system tailored for FinTech applications, are explained and discussed in this section. The evaluation focused on a comparative analysis of database architectures, specifically a PostgreSQL-only configuration using PostgreSQL versus a hybrid RDBMS+Redis configuration using PostgreSQL and Redis databases [6]. The database systems were subjected to varying load conditions, and performance metrics analysis using θ, T, and λ, as Table 1 illustrates.

Further, Table 1 compares the PostgreSQL-only and hybrid RDBMS+NoSQL architectures. In addition, it should be noted that θ, T, and λ requests in Table 1 refer to transaction creation (transfer) only and do not account for detail requests, which are issued in each case prior to the measurement period but are not included in the total requests above.

Furthermore, the current account balance is obtained internally as part of every transfer request and does not make up an additional request. Equal weights were assigned to the metrics $w1 = w2 = w3 = 1$ in Eq (3.1). The relative performance improvement is evaluated for all test cases using the results in Table 1, as explained in the following discussion, including Eqs (4.1)–(4.6).

To clarify, all the component parameters ($\theta 1$, $\theta 2$, $T1$, $T2$, $\lambda 1$, $\lambda 2$) and speedup ratios provided below are given in their raw units and would remain valid with respect to both formulations of the composite score provided below. The composite score ΔPk itself, on the other hand, is dependent upon the normalization used in Eq (3.1). A detailed comparison of the two forms of ΔPk for all three problems is provided in Table 3.

Table 3. Raw versus normalized composite score by test case.

Test case	θ_{norm} (%)	T_{norm} (%)	λ_{norm} (%)	ΔP_norm (equal weights)	ΔP_raw (equal weights, un-normalized)
Test 1	2.1	11.9	0.0	0.14	28.55
Test 2	3.2	5.2	0.0	0.08	11.97
Test 3	14.9	66.3	84.6	1.66	314.65

Test Case 1 (low load): θ1 = 2.34, θ2 = 2.39; $T1$ = 239.72, $T2$ = 211.22; and $\lambda 1 = 0$, $\lambda 2 = 0$

$$\Delta P1 = (2.39 - 2.34) + (239.72 - 211.22) + (0 - 0) = 28.55. \tag{4.1}$$

The speedup factor for the first test case is therefore

$$\text{Speedup1} = \frac{239.72}{211.22} \approx 1.13. \tag{4.2}$$

Test Case 2 (medium load): θ1 = 9.09, θ2 = 9.38; $T1$ = 224.00, $T2$ = 212.32; and λ1 = 0, λ2 = 0

$$\Delta P2 = (9.38 - 9.09) + (224.00 - 212.32) + (0 - 0) = 11.97. \tag{4.3}$$

The speedup factor for the second test case is therefore

$$\text{Speedup2} = \frac{224.00}{212.32} \approx 1.06. \tag{4.4}$$

Test Case 3 (high load): θ1 = 20.37, θ2 = 23.40; T1 = 469.99, T2 =158.48; and λ1 = 0.13, λ2 = 0.02

$$\Delta P3 = (23.40 - 20.37) + (469.99 - 158.48) + (0.13 - 0.02) = 314.65. \tag{4.5}$$

The speedup factor for the third test case is therefore

$$\text{Speedup3} = \frac{469.99}{158.48} \approx 2.97. \tag{4.6}$$

In Table 4, T is the average response time for both databases, and S represents the speedup factor for all three test cases.

Table 4. Speedup factor summary.

Test case	T_{RDBMS}	$T_{RDBMS+Redis}$	S
Test 1	239.72 ms	211.22 ms	**1.13**×
Test 2	224.00 ms	212.32 ms	**1.06**×
Test 3	469.99 ms	158.48 ms	**2.97**×

The HDB system achieves a 1.13× speedup in Test 1, 1.06× in Test 2, and a significant 2.97× speedup in Test 3. These results prove that while modest improvements occur under a normal load, the hybrid architecture drastically enhances performance under high-load conditions, as illustrated in Table 4, where the speedup factor comparison of RDBMS + Redis over the standalone RDBMS across three test cases confirms the HDB system's performance.

4.1. Effect of metric normalization on the composite score

The composite improvement in performance scores, ΔPk, is shown for each test case in Table 3 using both the raw or un-normalized difference values (ΔP_raw) and the normalized relative change equations given in Eqs (3.1)–(3.4) (ΔP_norm), together with the individual normalized contributions for each metric (θ_{norm}, T_{norm}, and λ_{norm}).

Further, it is noted that $\lambda 1 = 0.0$ in Test Cases 1 and 2, which means no failures at the RDBMS-only baseline. This comes as a result of λ_{norm} being defined as 0 through the ε-guard in Eq (3.4) rather than being undefined. There are two important observations from Table 3. First, ΔP_raw is measured in incomparable units, including the sum of change in RPS and the change in ms. It is also dominated numerically by the response time in each of the tests performed, which is clearly visible in Table 3. The response time metric provides 311.51 out of 314.65, or, in other words, about 99% of ΔP_raw, despite the response time being only one of the three metrics intended to be weighted in Test 3.

Furthermore, after normalization of each metric to a relative improvement scale, all three metrics contribute significantly to ΔP_norm. In Test 3, throughput, response time, and failure rate provide 14.9%, 66.3%, and 84.6%, respectively; thus, the latter is indicative of the fact that an improvement in the failure rate from 0.13% to 0.02% is a large reliability improvement, even though both figures are small in absolute terms. The normalized throughput and response time figures expressed in percentages in Table 3 are aligned with the initial results. ΔP is positive in all three tests for both formulations, raw and normalized. Hence, the qualitative observation that the HDB design outperforms the RDBMS-only one remains true for both formulations; the difference is that while the raw formulation is mostly dominated by the response time, the normalized one reflects a genuine, comparable combination of the three metrics.

Overall, the HDB system consistently delivers higher throughput and much lower response times across all test scenarios. Under a high load of 20.37 RPS, the PostgreSQL-only response time spikes to 469.99 ms, while the hybrid remains at 158.48 ms. Conversely, the hybrid cloud database reveals a 6.5-fold lower failure rate, 0.02% versus 0.13%, which proves it to be more scalable and reliable.

4.2. Explaining the throughput load relationship

Aggregate throughput (θ) increases in Tests 1–3, while the individual request response times do not decrease proportionally, as illustrated in Table 1. At first glance, this may seem confusing, because when a DBMS is overloaded, one would expect individual request response times to increase, and the system would handle fewer RPS. However, there are two related factors that explain this phenomenon.

First, in all three test cases, testing was performed under a closed workload with a specific number of simulated users, i.e., 5, 20, and 50, and zero think time at the client side; see Section 3.4. For a closed workload, the aggregate throughput of the database system is bounded by the number of simultaneous requests in flight, which, in turn, is capped by the number of concurrent users. The relation between the number of concurrent users and throughput can be expressed using Little's Law, namely $\theta \approx N / T$, where N is the number of concurrent users, and T is the mean response time per request [25]. In the case of Test 1, with $N = 5$, no more than five request–response pairs can be processed at any instance of time, regardless of the server's capacity, limiting the possible aggregate throughput significantly compared with the same server's performance under more concurrency conditions despite the similar request latency in Test 1 and Test 2. As N increases to 20 and then 50, more requests are processed concurrently, so the aggregate throughput increases, as shown in Table 1.

Second, instead of all users starting at once, Locust raises the number of users to the target number gradually during the first few seconds of every test cycle. This means that during Test 1, since there were only 300 requests, the time used in raising the user numbers is a relatively large percentage of the total test time, making the mean throughput artificially low compared with Test 3, where the same ramp-up time would be distributed among 5000 requests.

Further, under baseline and medium-load conditions, the hybrid cloud database architecture was shown to make marginal consistent improvements. It achieved a higher throughput, with 2.39 and 9.38 RPS versus 2.34 and 9.09 RPS for the RDBMS-only system, and lower average response times in both configurations, maintaining 100% reliability with zero failures [6]. The superiority of the HDB model became profoundly evident under high load, as illustrated in Figure 4. Also, the RDBMS-only system experienced a severe degradation in performance, with the average response time increasing to 469.99 ms, as illustrated in Figure 5.

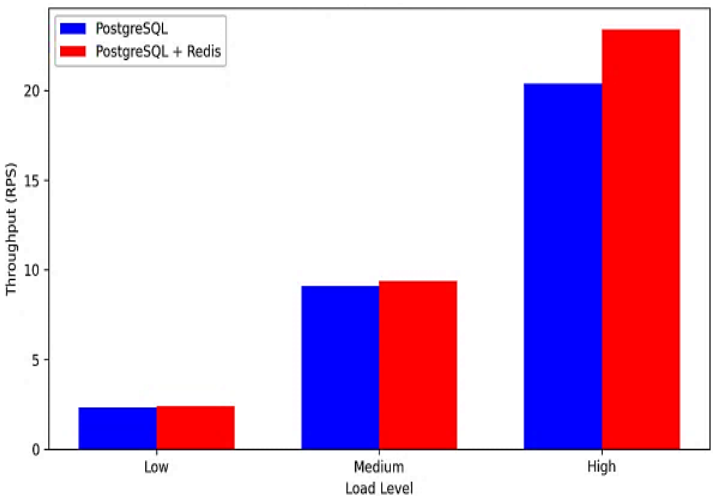


Figure 4. Throughput vs. Load.

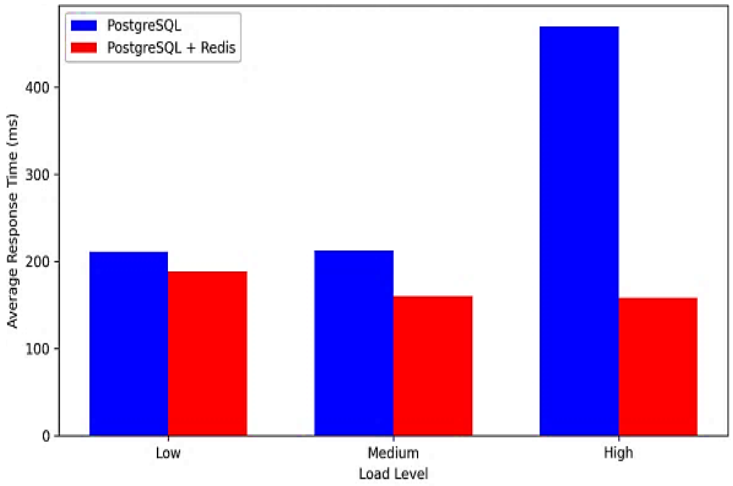


Figure 5. Response time vs. Load.

Conversely, the hybrid cloud database architecture maintained a responsive state by achieving an average latency of 158.48 ms under high load, with a corresponding speedup factor of nearly 3× [6]. This improved performance was achieved while still sustaining a higher throughput of 23.4 RPS against 20.37 RPS for the RDBMS-only setup, with a significantly lower failure rate of 0.02% against 0.13% for the RDBMS-only setup, as shown in Figure 6.

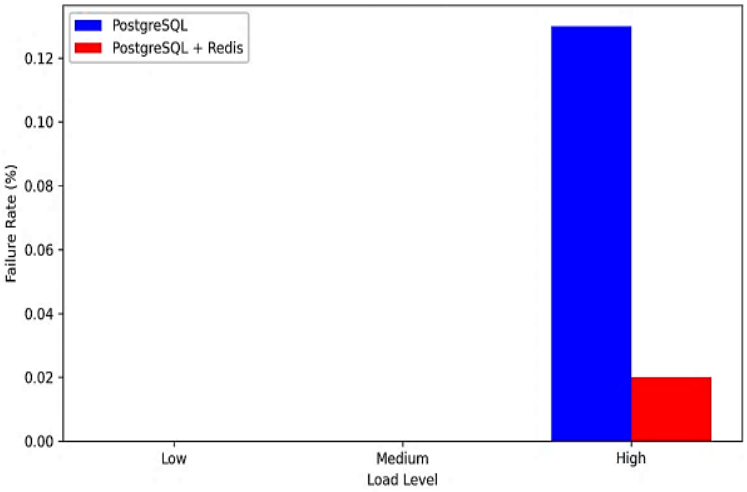


Figure 6. Failure rate vs. Load.

Furthermore, the research experiments prove that for high-volume transaction levels, the hybrid RDBMS + Redis architecture consistently outperforms the classical PostgreSQL-only equivalent. As seen from the achieved speedup and performance improvements, given by $\Delta \boldsymbol{Pk} > 0$ and a speedup of $\boldsymbol{k} > 1$ for all test cases, adding an in-memory layer to the database system increases not only its reliability but also its performance. It can be confirmed that database access delay becomes a critical component affecting the system's performance in microservice FinTech platforms and overcoming it through an HDB architecture improves performance levels significantly. Performance charts from the Locust dashboard of both configurations under high load are shown in Figures 7 and 8.

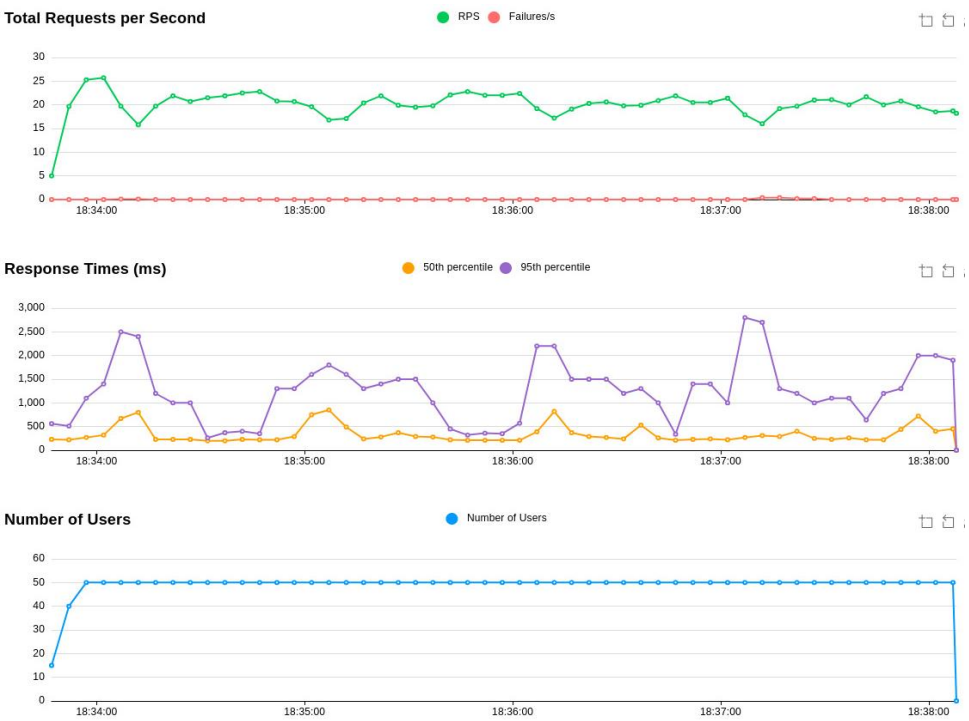


Figure 7. Locust performance charts: RDBMS-only.

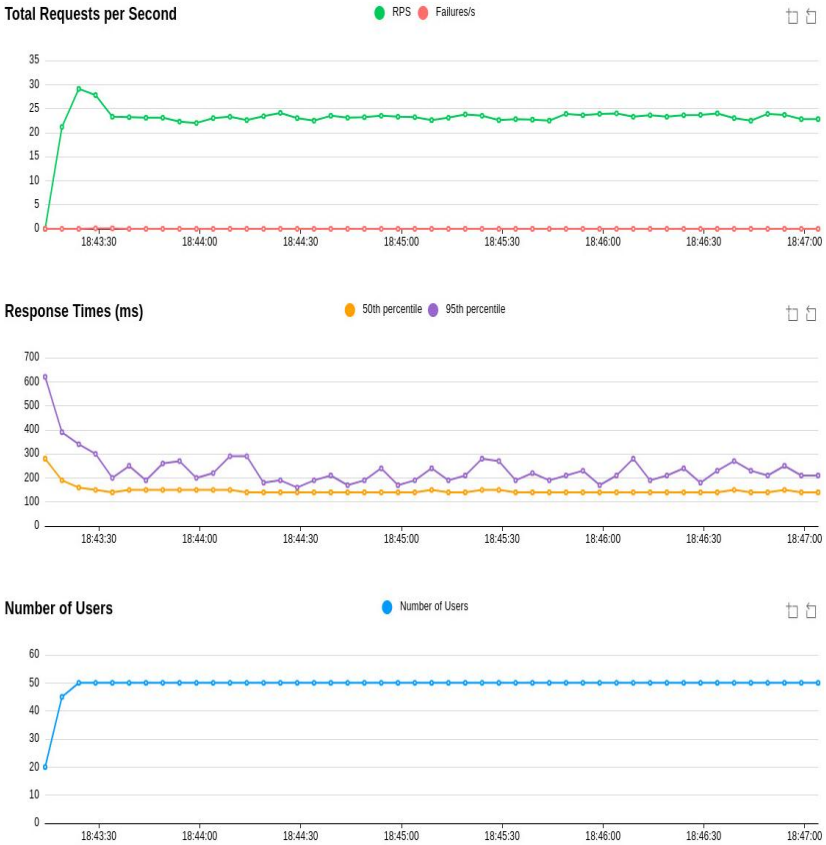


Figure 8. Locust performance charts: Hybrid.

Lastly, the RDBMS-only database configuration in Figure 7 exhibits fluctuating Locust performance under high load, as seen in the unstable RPS and response time spikes. However, Figure 8 exhibits the hybrid cloud database's steady performance under high load, confirming its ability to maintain appropriate throughput and low latency.

4.3. Tail-latency structure under high load

The values of T given in Table 1 are averages; for Test 3 in particular, they conceal a right-skewed distribution of the response times, which contains information beyond what the average reveals. More information regarding the distributions and endpoint-wise data is explained and discussed here (Tables 4 and 5). This is limited to Test 3 only, rather than provided for all three test scenarios, since Table 1 already indicates 0% failures and very similar averages of response times for both configurations in Tests 1 and 2, offering little or no details about the tail-driven failure-related performance behavior that warrants an analysis of the underlying distribution.

A further investigation into the Locust time series outputs underlying Figures 7 and 8, with the 95th and 50th percentiles of response time, sampled roughly every 5 s during the entire duration of the Test 3 run, reveals that both setups feature a repeated cycle of latency spikes for the 95th percentile occurring every 60–90 s, rather than a steady increase in the latency of all requests' response times. In the case of the RDBMS-only setup, these spikes measure around 1800–2700 ms with a 50th percentile range of 200–800 ms between spikes, whereas in the case of the hybrid setup, a similar periodicity can be observed, but the spikes are much lower, at about 250–300 ms with a stable 50th percentile value of 130–150 ms. This is summarized in Table 3, with evidence.

Table 5. Test 3, with 50 concurrent users: Percentile evidence.

Configuration	p50 (ms)	p95 peak (ms)	Approximate spike periodicity	Reported mean (ms)
RDBMS-only	≈ 200–800	≈ 1800–2700	≈ 60–90 s	469.99
Hybrid	≈ 130–150	≈ 250–300	≈ 60–90 s	158.48

The condensed statistics associated with the transfer endpoint itself for Test 3 provide additional validation to the percentile analysis above. The RDBMS-only configuration, where transfers made 5,000 requests, had seven failures, i.e., 0.14%, which is similar to 0.13% (see Table 1), with an average response time of 469.99 ms, a minimum of 140 ms, and a maximum of 3162 ms. The hybrid system, where transfers made 5000 requests, had one failure, i.e., 0.02%, which is comparable to Table 1, with an average response time of 158.48 ms, a minimum of 104 ms, and a maximum of 1007 ms. It is noteworthy that the ratio between the maximum response times for both configurations, i.e., 3162 ms/1007 ms ≈ 3.14, compares well with the ratio between the average response times, i.e., the speedup factor in Eq (4.6), with ≈ 2.97, suggesting that the HDB configuration's benefits hold consistently throughout the response times' range, including extremes.

4.4. Estimating the standard deviation from summary statistics

The total requests, mean, minimum, and maximum values are available for the transfer endpoint

(Table 2). Two standard three-point estimation procedures that can be used where the n, minimum (a), maximum (b), and mean (μ) values are known were considered to determine the underlying spread.

First, the method of moments was applied in order to fit a triangle distribution, where the study used $\mu = (a + b + c)/3$ to estimate the implied mode, where $c = 3\mu - a - b$; thereafter, $\sigma^2 = (a^2 + b^2 + c^2 - ab - ac - bc)/18$ was calculated. For the RDBMS-only configuration in Test 3, a = 140, b = 3162, and μ = 469.99. This implied mode was calculated to be $c \approx -1{,}892$ ms. For the hybrid system in Test 3, a = 104, b = 1007, and μ = 158.48. This was calculated to be $c \approx -636$ ms. Both estimated values of the mode cannot be used since they are not within [a, b] and therefore are impossible for a triangle distribution.

On the other hand, this research finding is valuable, since it clearly shows that the response times' distribution is too sharply concentrated at the minimum, with an extremely low upper tail for the given data to be fitted by a triangle (single-mode, linear tapering) distribution, which results in quantitative evidence of the rare and periodic contention events for tail latency. As a fallback method that does not involve calculating the mode, the program evaluation and review technique (PERT), beta-PERT, approximation of the three-point standard deviation formula, $\sigma \approx \sqrt{[(\mu - a)(b - \mu)/7]}$, provides a standard deviation of $\sigma \approx 356$ ms for the RDBMS-only system and $\sigma \approx 81$ ms for the hybrid equivalent. A cruder range-based standard deviation approximation formula, $\sigma \approx (b - a)/6$, is typically used as a first-order approximation technique. This leads to $\sigma \approx 504$ ms for the RDBMS-only configuration and $\sigma \approx 151$ ms for the hybrid one. The difference between these two formulas is significant due to the highly skewed nature of the distribution being considered and violates the assumptions of both techniques, which led to their use as order-of-magnitude approximations only; see Table 6.

Lastly, although these disagree in magnitude, they are consistent in direction, where the implied coefficient of variation for PostgreSQL-only is larger than for the hybrid configuration using both approaches with 0.76 versus 0.51 using PERT and 1.07 versus 0.95 using the range approach. This means the response times for RDBMS-only vary proportionately more relative to their own mean than those of the HDB system, as one would expect, given its increased exposure to the tail latency effects found from the percentiles, as shown in Table 5, and endpoint min/max, as shown in Table 2. Thus, it has been proven that these are estimated, with model-based values based on the summary statistics.

Table 6. Test 3: Approximate standard deviation estimates from summary statistics.

Configuration	σ (PERT approx., ms)	σ (range/6 approx., ms)	CoV = σ/μ (PERT)	CoV = σ/μ (range/6)
RDBMS-only	≈ 356	≈ 504	≈ 0.76	≈ 1.07
Hybrid	≈ 81	≈ 151	≈ 0.51	≈ 0.95

4.5. Failure analysis from statistics

The periodicity recurs and persists in both database systems, which indicates that there is a common cause on the PostgreSQL side rather than this being a result of the HDB system's design; more specifically, the writing or autovacuum operations for the account and transactions tables in the PostgreSQL database server, which happen on a regular fixed-interval basis largely independently of the client workload in the default PostgreSQL configuration [26,27]. The HDB system does not eliminate this periodic event; instead, because a large number of the queries for account details in the HDB are retrieved via Redis and not PostgreSQL, where fewer in-flight requests are caught up in the

event in PostgreSQL, this results in significantly less of a spike in latency than in the PostgreSQL-only database system during the same event.

Further, the causes of failure for the RDBMS-only Test 3 in Table 1 are not client-side timeouts of unknown origin. Reviewing the Locust failure log in connection with the RDBMS-only Test 3 shows that there were seven failures on the transfer endpoint, each responding with an HTTP 500 status code, with the error message "QueuePool limit of size 10 overflow 10 reached, connection timed out, timeout 30.00", as shown in Figure 9. This is consistent with the 0.13% failure rate seen in Table 1, i.e., 0.13% × 5000 ≈ 6.5, with seven failures. The message in Figure 9 is related to SQLAlchemy's own error reporting and directly pinpoints the problem as being within the database engine, as the transaction service was set to have a base pool size of 10, with an additional maximum overflow of 10. In other words, at most 20 concurrent database connections can be made to the PostgreSQL database, which falls far short of the 50 concurrent users in Test 3. Once the ceiling is hit, additional requests are queued up for up to the pool_timeout of 30 s before SQLAlchemy raises a timeout error, with an HTTP 500 status.

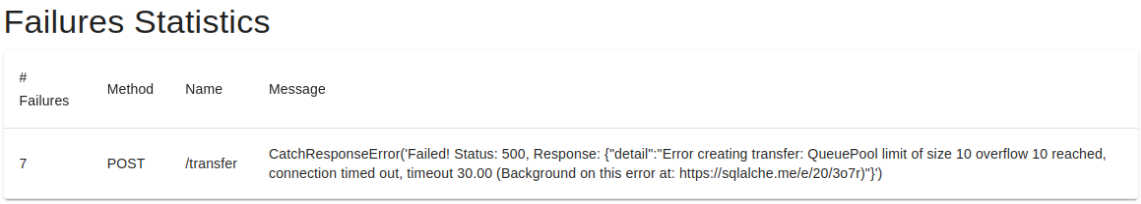


Figure 9. Locust-reported QueuePool exhaustion error, showing the Test 3 failure rate of RDBMS-only.

Furthermore, given the ceiling of 20 concurrent connections in the pool, the 20 concurrent users from Test 2 are on par with the capacity limit in the pool rather than above it, and this is consistent with the 0% failure rate indicated in Table 1 for Test 2, since, at such a concurrency level, the actual demand for simultaneous database connections cannot quite reach the ceiling, because each query takes little time relative to the rate of requests. In contrast, the 50 concurrent users of Test 3 amount to 2.5 times the capacity ceiling, making it significantly more probable that the simultaneous demand exceeds the capacity, consistent with the seven recorded failures, due to the periodic contention windows on the PostgreSQL database side. This also proves that the connection pool limit configuration (20), which was set up, was not undersized relative to Test 2's full concurrency, but was insufficient to ensure no failures under Test 3's increased concurrency levels, as expected from the considerably higher peak demand, on the order of 50 or more connections, obtained from the calculation of independent spike loading. The difference in reliability between the two configurations can be attributed to the decrease in the connection hold time of transactions due to the use of an HDB architecture, rather than just the size of the connection pool itself.

Similarly, the application of this connection pool exhaustion analysis technique in a comparative manner provides a quantitative explanation as to why the failure rate of Test 3 in the HDB setup, with 0.02%, is much lower than the failure rate in the PostgreSQL-only setup, with 0.13%, by a factor of 6.5×, instead of this observation being regarded as separate and not explained. The application of Little's Law using Test 3's throughputs and mean response times, as shown in Table 1, gives the average concurrent PostgreSQL connection demand of about 9.6 connections in the

PostgreSQL-only setup (20.37 RPS × 469.99 ms) and about 3.7 connections in the hybrid setup (23.40 RPS × 158.48 ms), which is a 2.6-fold difference. With respect to the same nominal limit of a connection pool ceiling of 20, it is evident that PostgreSQL-only runs at about 48% ceiling utilization, while HDB, PostgreSQL+Redis, operates at around 19%.

In contrast, the reduced PostgreSQL hold time for each transaction in the hybrid database configuration (a result of reading the account balances from Redis instead of PostgreSQL) allows much more room before any contention event would raise the instantaneous load above the threshold of the pool capacity. This leads to the former difference in failure rates (6.5×) being higher than the latter in utilization (2.6×), following the known nonlinear connection between utilization and the blocking probability close to a queueing system's saturation point, where a small discrepancy in average utilization results in a significant difference in the tail. It is not surprising that the failure rate gap is larger, with 6.5×, than the average utilization gap, i.e., 2.6×, as this is in accordance with the well-established nonlinearity in the relationship between utilization and blocking probability around the capacity limit of a queuing system. A small difference in average utilization results in a large difference in tail behavior, which serves as an explanation for the improvement in performance through the use of the cache in-memory strategy. This explanation is pointed out as strongly evidence by the existing Locust telemetry, although it was not validated using direct instrumentation on the database side, such as PostgreSQL checkpoints or CloudWatch input/output (I/O) and connection count statistics for the test window.

Overall, the findings indicate that architectural benefits associated with caching stand out more as the workload becomes higher, although there are modest improvements in the database systems' performance when the load is low or medium. This highlights the importance of database design systems not only for peak conditions but also for normal loads. Lastly, given the results obtained from the experiments, it is recommended for FinTech cloud microservices to adopt HDBs that utilize the dual benefits of relational databases for ACID compliance and in-memory caching for improved performance. This HDB architecture becomes beneficial when considering frequently queried data, such as account details. The testing and designing of such architectures should utilize performance evaluation approaches, which consider aspects such as speedup, latency, failure rate, and throughput.

5. Conclusions

The research findings show that applying RDBMS and NoSQL cache improves the performance of the microservice used to process financial transactions. From all the tests conducted, the HDB configuration performed better than the RDBMS-only baseline in scalability, latency, and reliability especially under high-load conditions, where a 2.97× speedup, 66.3% reduced latency from 469.99 ms to 158.48 ms, and a 6.5× reduction from 0.13% to 0.02% in the failure rate were achieved.

Indeed, the introduction of Redis as a caching layer for account details relieved PostgreSQL from excessive load so that the relational database could concentrate on transactional consistency and writing operations. The procedure as described in Section 4.5; in particular, by decreasing the time a database connection was retained by each transaction, the caching tier managed to lower the connection requests below the ceiling of the connection pool. This improves the throughput and responsiveness, which results in a better financial system's performance.

Further, this research study also reinforces the notion that polyglot persistence, which involves combining relational and non-relational databases to capitalize on the strengths of each type, can boost

the performance of FinTech applications. The hybrid setup proved to be more efficient in terms of response times, scalability in the presence of concurrent users, and better handling of load spikes. This is consistent with the results found in previous comparative research studies but extends the research by offering a performance model and a root cause analysis suited for FinTech money transactions [3,4]. Hence, the proposed architectural design acts as an implementation guide for designers looking to build highly efficient, responsive, and scalable FinTech database systems. However, the research study is limited to performance metrics, throughput, response time, and failure rate, and does not account for the cost implications or long-term operational aspects, such as cache invalidation or consistency management. Indeed, future research work should focus on performance benefits in relation to cost, adaptivity for caching and scaling, as well as more testing of NoSQL databases to generalize results and enhance HDBs' performance in financial environments.

Furthermore, the research examined all combinations co-located in one VPC of the AWS environment, where the application tier was applied using Amazon's ECS tasks, the PostgreSQL database being provisioned with Amazon's RDBMS, and the cache being provisioned with AWS ElastiCache for Redis, as a controlled variable; the performance of a distributed multinode topology was not assessed in this study. Although pool exhaustion itself has been validated to be the proximate cause of the PostgreSQL-only configuration's performance failures through the corresponding diagnostic error message, the periodic tail latency issue and its possible relationship to the PostgreSQL checkpoints and autovacuums, as factors increasing the connection holding times. Thus, the possibility of pool exhaustion, have not been verified by direct instrumentation of the database, as log information was not obtained from the initial test runs and should be considered as the most probable explanation supported by the available evidence. Finally, the analysis of this research study included only the performance characteristics measured at the application level, namely the throughput, latency, and failure rates, without assessing the server-side resource usage metrics, such as central processing unit (CPU), memory, disk I/O usage, or concurrent database connection pool size over time during the tests.

A noticeable limitation is related to the cache parameter attribution and financial caching risk profile. The study only examined one type of hybrid caching against the PostgreSQL-only relational database without varying, independently, the values of the cache parameters, such as the hit rate, invalidation strategy, and consistency window. Despite the fact that the research findings prove that the benefits are attributed to the reduced demand for PostgreSQL connections due to Redis handling the read queries, the effect of particular cache parameters was not isolated. Therefore, further research may involve performing experiments where such values and parameters are varied independently. Another research issue is that the study did not pay attention to the risks related to caching in financial systems that may cause outdated information in the consistency window between PostgreSQL's commitment and Redis updates, cache failure due to multiple requests when there is a cache miss, and cache avalanche due to mass key expiration or the failure of Redis. In order to prove that the described database system is ready for production use in the financial domain, further research is needed.

Moreover, three notable contributions were made in this research study. First, it recommends an HDB system architecture for FinTech cloud microservices, showing how relational and nonrelational database systems can be integrated effectively. Second, it provides a formal mathematical model for performance evaluation that enables rigorous and repeatable comparisons of database systems using performance metrics. Finally, the study provides practical insights into behavior under specific loads through data collected from performance testing. Nevertheless, this research does not cover the economic or operational aspects of the HDB architecture. Rather, it is confined to measures such as

throughput, response time, and failure rate for performance evaluation. Specifically, parameters such as data consistency assurances, cache invalidation strategies, and the costs associated with maintaining an HDB architecture are not covered in the study. These factors are important for real-world applications and may influence the feasibility of the proposed HDB model and FinTech solution.

In further studies, there are several directions to expand this research study. First, to understand the trade-offs between better performance and costs for cloud infrastructures, a cost–performance analysis could be performed. Second, exploring complex caching techniques and consistent updates could make more improvements to the database system's performance. Third, incorporating workload-aware resource management and dynamic scalability techniques in the case of fluctuating load is another direction for future work. Fourth, an assessment of the performance characteristics of the HDB architecture under a distributed topology when the PostgreSQL and Redis databases are situated in separate physical hosts or availability zones is needed to find out whether the positive effects of the cache affect the findings of this research hold or not in non-colocation scenarios. Fifth, another research question considers the assessment of the existing caching solution in terms of its efficiency compared with scalable alternatives, such as read replicas or sharding. This could be compared against a PostgreSQL-based database only, while the current research compares the caching solution with a single RDBMS-node database setup, but does not provide any insight into how it performs in contrast to any other scaling techniques for the relational tier. Sixth, conducting load tests of the database system and getting diagnostic data from the database (PostgreSQL checkpoint logs, pg_stat_bgwriter, connection pools statistics, and input/output operations per second (IOPS)/CPU metrics from AWS CloudWatch) to test the checkpoint/autovacuum hypothesis related to the peaks of tail latencies.

In addition, extending the scope of evaluation from just application-level metrics to server-side metrics, including CPU, memory, disk I/O, and connection pool usage. Also, since the configured maximum size of the connection pool was not enough to prevent failures in Test 3 (pool size = 10, max overflow = 10), a study could be possible to gather metrics on how many concurrent connections are held at the peak time; for example, by using instrumentation logging of the pool or pg_stat_activity. This determines the actual maximum load and determines if a much higher value of the maximum size of the connection pool of 50 and above, according to the estimated load-independent spike, is enough or if reducing the connection hold times is necessary.

Lastly, other types of NoSQL databases, such as distributed key value stores and document stores, could be used for generalizing new research findings obtained from this research study to different FinTech database system scenarios. Taking all these into account, the proposed hybrid cloud database architecture provides a solid foundation for developing highly efficient FinTech applications, and the contributions to knowledge gained from the research findings provide useful guidance for practitioners and researchers on how to develop such database systems to enhance FinTech systems' performance.

Use of AI tools declaration

The authors declare that they have not used artificial intelligence (AI) tools in the creation of this article.

Acknowledgments

The authors would like to thank the Department of Computer Science and Digital Technologies (CDT)

staff at the University of East London for their help and support, especially Dr. Aaron Kans and Dr. Mustansar Ghazanfar for their guidance, dedication, and professionalism.

Conflict of interest

The authors declare that there is no conflict of interest.

References

1. G. Kou, Y. Lu, FinTech: A literature review of emerging financial technologies and applications, *Financ. Innovation*, **11** (2025), 1. https://doi.org/10.1186/s40854-024-00668-6
2. J. Thönes, Microservices, *IEEE software*, **32** (2015), 116. https://doi.org/10.1109/MS.2015.11
3. M. Soni, V. Jyotinagar, SQL vs NoSQL databases for microservices: A comparative survey, in *2023 2nd International Conference on Edge Computing and Applications (ICECAA)*, (2023), 17–22. https://doi.org/10.1109/ICECAA58104.2023.10212190
4. R. Laigner, Y. Zhou, M. A. Vaz Salles, Y. Liu, M. Kalinowski, Data management in microservices: State of the practice, challenges, and research directions, in *Proceedings of the VLDB Endowment*, **14** (2021), 3348–3361. https://doi.org/10.14778/3484224.3484232
5. D. Mahajan, C. Blakeney, Z. Zong, Improving the energy efficiency of relational and NoSQL databases via query optimizations, *Sustainable Comput. Inf. Syst.*, **22** (2019), 120–133. https://doi.org/10.1016/j.suscom.2019.01.017
6. H. AbouGrad, E. Iyahen, S. Alsaid, Cloud-based hybrid-database architecture: Optimising FinTech real-time AWS elastic container microservices database system performance, in *6th International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC-26)*, (2026), 1–6.
7. N. Pantelic, L. Matic, L. Jakovljevic, S. Eric, M. Eric, M. Stefanović, et al., Benchmarking SQL and NoSQL persistence in microservices under variable workloads, *Future Internet*, **18** (2026), 53. https://doi.org/10.3390/fi18010053
8. R. Cattell, Scalable SQL and NoSQL data stores, *ACM SIGMOD Record*, **39** (2011), 12–27. https://doi.org/10.1145/1978915.1978919
9. R. Hecht, S. Jablonski, NoSQL evaluation: A use case oriented survey, in *2011 International Conference on Cloud and Service Computing*, (2011), 336–341. https://doi.org/10.1109/CSC.2011.6138544
10. S. Ionescu, V. Diaconita, A. Radu, Engineering sustainable data architectures for modern financial institutions, *Electronics*, **14** (2025), 1650. https://doi.org/10.3390/electronics14081650
11. A. A. E. Alflahi, M. A. Y. Mohammed, A. Alsammani, A scalable transaction management framework for consistent document-oriented NoSQL databases, *Inf. Syst.*, **140** (2026), 102719. https://doi.org/10.1016/j.is.2026.102719
12. A. Rao, D. Khankhoje, U. Namdev, C. Bhadane, D. Dongre, Insights into NoSQL databases using financial data: A comparative analysis, *Proc. Comput. Sci.*, **215** (2022), 8–23. https://doi.org/10.1016/j.procs.2022.12.002
13. D. Jawale, S. Shelke, S. Yadav, Analyzing the impact of database architecture on performance: SQL vs NoSQL, *Int. J. Math. Comput. Res.*, **14** (2026), 61–65. https://doi.org/10.47191/ijmcr/v14iSPC3.13

14. A. E. Topcu, A. M. Rmis, Y. I. Alzoubi, A. O. Çibikdiken, D. Chowdhury, E. Elbasi, et al., Evaluating the performance of NoSQL databases for big data in cloud computing environments, *HighTech Innovation J.*, **6** (2025), 808–830. https://doi.org/10.28991/HIJ-2025-06-03-05
15. H. AbouGrad, A. Shabarshov, M. Yegamati, B. A. Romo, M. S. Sharif, Cloud databased contactless mobile payment system framework: An asymmetric encryption secured financial transactions model, *Sci. J. Res. Rev.*, **4** (2026), 1–16. https://doi.org/10.33552/SJRR.2026.04.000600
16. A. K. Bayya, Data-driven predictive analytics and decision- making in fintech using MongoDB and high-throughput data pipelines, *Int. J. Algorithms Design Anal. Rev.*, **3** (2025).
17. N. Bagam, Implementing scalable data architecture for financial institutions, *Stallion J. Multidiscip. Assoc. Res. Stud.*, **2** (2023), 27–40. https://doi.org/10.55544/sjmars.2.3.5
18. S. Ionescu, V. Diaconita, Transforming financial decision-making: The interplay of AI, cloud computing and advanced data management technologies, *Int. J. Comput. Commun. Control*, **18** (2023), 5735. https://doi.org/10.15837/ijccc.2023.6.5735
19. D. T. Absari, A. Djunaidy, T. D. Susanto, Design science research methodology concept and its application in the development of online community communication system for Selotapak Village, in *2022 9th International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE)*, (2022), 236–241. https://doi.org/10.1109/ICITACEE55701.2022.9924092
20. Locust, Locust Documentation, 2026. Available from: https://docs.locust.io/.
21. A. T. Kabakus, R. Kara, A performance evaluation of in-memory databases, *J. King Saud Univ. Comput. Inf. Sci.*, **29** (2017), 520–525. https://doi.org/10.1016/j.jksuci.2016.06.007
22. Amazon Web Services, What is Amazon Elastic Container Service? 2026. Available from: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html.
23. Amazon Web Service, What is Amazon VPC? 2026. Available from: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html.
24. Amazon Web Services, What is Amazon ElastiCache? 2026. Available from: https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/WhatIs.html.
25. J. D. C. Little, A proof for the queuing formula: L = λW, *Oper. Res.*, **9** (1961), 383–387. https://doi.org/10.1287/opre.9.3.383
26. The PostgreSQL Global Development Group, WAL Configuration, 2026. Available from: https://www.postgresql.org/docs/current/wal-configuration.html.
27. The PostgreSQL Global Development Group, Routine Vacuuming, 2026. Available from: https://www.postgresql.org/docs/current/routine-vacuuming.html#AUTOVACUUM.



©2026 the Author(s), licensee AIMS Press. This is an open access article distributed under the terms of the Creative Commons Attribution License (https://creativecommons.org/licenses/by/4.0)