eG Innovations

Beyond Cloud Monitoring:

Eight Lessons for Delivering High-Performance Cloud Applications

A Real-World Case Study

www.eginnovations.com

01 Introduction

Over the past decade, cloud computing has transformed the way enterprises build, deploy, and scale their digital ecosystems. From startups to Fortune 500 giants, organizations have embraced the cloud for its promise of agility, elasticity, and reduced operational overhead. Cloud spending now represents a dominant share of IT budgets, with many companies pursuing hybrid, cloud-first or even cloud-only strategies as they phase out or reduce reliance on traditional data centers.

Hyper-scalers such as Amazon AWS, Microsoft Azure, Google Cloud Platform (GCP), and Oracle Cloud Infrastructure (OCI) offer a staggering array of managed services that accelerate innovation — compute, storage, databases, analytics, AI, and serverless technologies. These services simplify infrastructure management and make it easier than ever to scale applications.

Yet, for all the benefits that cloud computing brings, performance management and observability remain as complex — and as critical — as ever. Many organizations assume that migrating to the cloud automatically solves their visibility challenges. Others rely exclusively on native monitoring tools such as Amazon CloudWatch, Azure Monitor, or Google Operations Suite, believing these provide sufficient visibility.

The reality is very different. Cloud observability is a shared responsibility. Cloud providers ensure the health of their infrastructure — but the availability, performance, and user experience of your applications remain your responsibility. When transactions slow down or outages occur, users don't raise tickets with the cloud service provider; they call your helpdesk. The resulting productivity loss, customer dissatisfaction, and reputational impact fall squarely on your organization, not your cloud service provider.

Achieving true observability in the cloud requires visibility that cuts across every layer — from user experience and application logic to cloud infrastructure and managed services. It demands correlation and interpretation (not just collection) of metrics.

This whitepaper presents our experiences of working with a large manufacturing and retail company that was in the process of scaling their application in the cloud. In this real-world case study, an enterprise ERP (Enterprise Resource Planning) system hosted on AWS was being scaled to support additional workloads. We have documented our experience in assisting this deployment, where a sudden performance degradation after a major scale-up revealed how elusive cloud-era bottlenecks can be. Through this experience, we uncover eight critical lessons that every IT, DevOps, and SRE team should take away to achieve consistent, high-performance operations in the cloud.

This case study illustrates not only the technical expertise required to troubleshoot modern multi-tier applications but also the importance of full-stack, end-to-end observability enabled by tools like eG Enterprise to correlate performance data across applications, databases, operating systems, and cloud services and pinpoint the true root-cause quickly and conclusively.

02 Application Deployment on AWS Cloud

The application being scaled was a two-tier ERP solution, with the front-end using Java technologies. The application was initially deployed in an on-premises data center and later migrated to AWS cloud. The ERP functionality provided by the application was accessed by staff in a few thousand retail stores. The architecture of the application in AWS is shown in Figure 1.

Users accessed the application from browsers on store desktops through secure HTTP/HTTPS connections. The requests were routed via a Network Load Balancer (NLB) and Application Load Balancer (ALB) combination.

- NLB ensured high availability and can handle millions of requests per second with ultra-low latency, making it ideal for distributing incoming traffic across multiple Availability Zones.

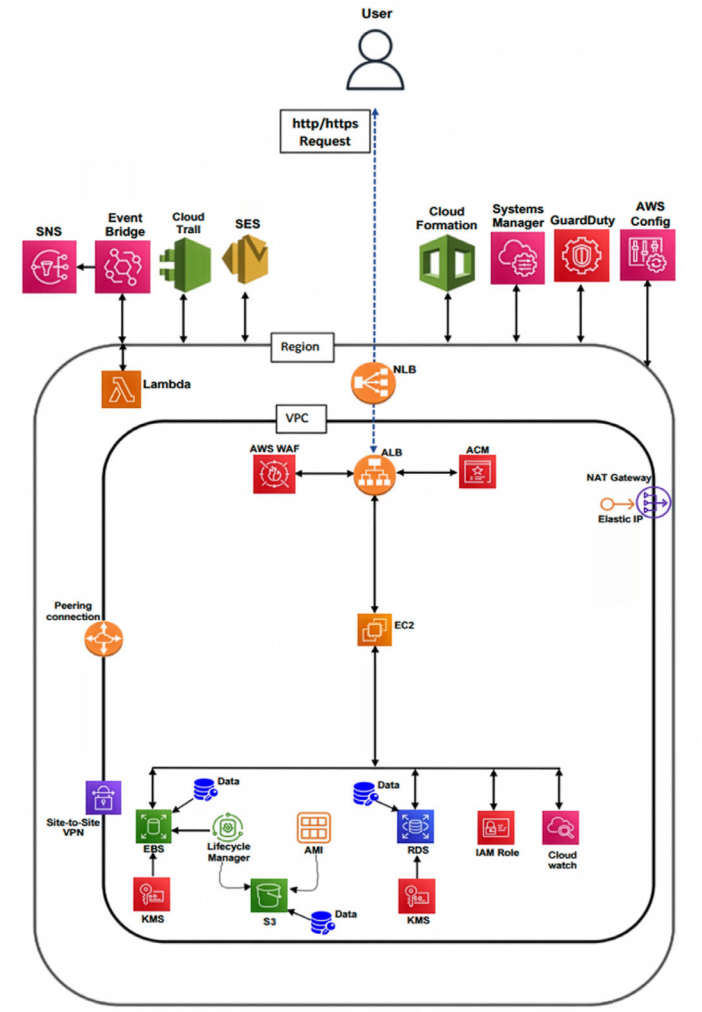


Figure 1: The ERP application architecture on AWS.

- ALB operates at Layer 7, allowing policies and routing rules to be enforced for different application paths, user groups, and microservices.

The ALB was integrated with AWS Web Application Firewall (WAF), which provided protection from common web exploits such as SQL injection, cross-site scripting (XSS), and distributed denial-of-service (DDoS) attacks. AWS Certificate Manager (ACM) handled SSL/TLS certificates to secure communications between the user and application endpoints.

Behind the ALB, Amazon EC2 instances hosted the Java-based ERP web tier. These EC2 instances were deployed within a Virtual Private Cloud (VPC) for isolation and secure connectivity. Outbound internet access from private subnets were managed through a NAT Gateway associated with an Elastic IP.

The backend database tier used Amazon RDS for Microsoft SQL Server, providing a managed, scalable, and fault-tolerant relational database environment. Data storage by the RDS instance was encrypted using AWS Key Management Service (KMS), ensuring data security at rest. Application data was further backed up and stored in Amazon S3, which provides durable and cost-effective storage.

For persistent block storage, Elastic Block Store (EBS) volumes were used, managed by AWS Data Lifecycle Manager to automate backup creation and retention policies.

Terraform was employed to automate the provisioning and management of the entire infrastructure stack — from VPC creation and EC2 deployment to security group configuration and RDS instance setup. This Infrastructure-as-Code (IaC) approach ensured consistency, repeatability, and easy rollback during upgrades or scaling operations.

AWS Services Used	Purpose
Amazon Machine Image (AMI) service	For creation of standardized, version-controlled images of the EC2 instances, simplifying recovery and deployment
Amazon Simple Email Service (SES)	To send email notifications directly from the ERP application to end users for alerts, invoices, or workflow updates
AWS Lambda and EventBridge	To enable serverless automation for tasks such as event-driven workflows, log analysis, or alert triggers
AWS CloudTrail	For governance, compliance, and audit visibility by logging all API activities
AWS Systems Manager	For simplified instance management, patching, and configuration control
AWS Guard Duty	To monitor for threats and anomalous activity, enhancing the security posture of the environment
AWS Config	To track configuration changes across all AWS resources to ensure compliance with internal and external standards
Site-to-Site VPN and VPC Peering	For secure connectivity to on-premises systems or other cloud environments, ensuring hybrid interoperability
AWS CloudWatch	For performance and operational visibility, integrated with eG Enterprise

Table 1: Details of AWS services used (besides EC2, EBS and RDS).

03 | Scaling the Application

When the ERP application was first migrated to AWS, it was sized appropriately for the business' operational load at that time. The initial configuration consisted of an Amazon EC2 instance with 16 vCPUs and 64 GB of RAM serving the Java-based web tier, and an Amazon RDS instance with a similar configuration for the backend database. The backend ran on Microsoft SQL Server Web Edition, chosen for its cost-effectiveness and licensing simplicity.

Users from over 3,000 geographically distributed retail stores accessed the application through the cloud-based web front-end. For more than a year following the migration, the system performed reliably — response times were steady, resource utilization was balanced, and users experienced consistent application availability across regions.

However, following a series of strategic acquisitions, the business had a requirement to onboard two newly acquired subsidiaries into the same ERP environment. This expansion would increase the total number of active store users to over 10,000 locations worldwide — representing more than a three-fold increase in workload.

While the cloud environment provided elasticity and scalability, the IT team recognized that simply adding more users without adjusting resource capacity and configurations would lead to potential performance degradation. To prepare for this major scale-up, they initiated a detailed benchmarking and capacity planning exercise using eG Enterprise, the organization's unified observability platform.

Using eG Enterprise's end-to-end performance visibility, the IT team gathered detailed insights into:

- Application response times under varying transaction volumes.
- Database query latencies and buffer pool utilization.
- Thread pool and connection pool behavior at peak loads.
- Resource consumption patterns (CPU, memory, and I/O) across EC2 and RDS layers.

By analyzing these metrics, the team identified two critical upgrade requirements to handle the increase in workload:

1. **Database Upgrade:** The existing SQL Server Web Edition imposed resource and feature limitations that would restrict scalability under the expected transaction volume. Upgrading to SQL Server Standard Edition was necessary to leverage advanced performance features (such as enhanced parallelism, in-memory optimizations, and higher resource thresholds).
2. **Infrastructure Scaling:** To sustain the higher workload and maintain existing performance baselines, both the EC2 web tier and RDS database tier needed to be upgraded to instances with double the CPU and memory capacity. This increase in compute power and memory would ensure sufficient headroom for connection pooling, caching, and concurrent request handling.

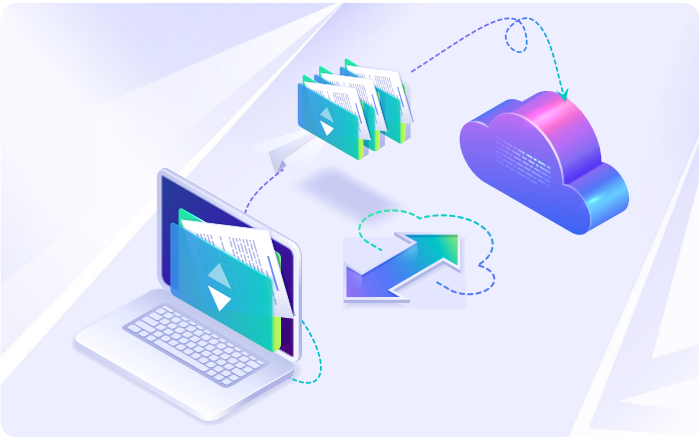
Through this proactive assessment, the organization was able to define a clear scaling strategy grounded in empirical data rather than assumptions. The benchmarking insights from eG Enterprise provided confidence that the upgraded configuration would meet both current and anticipated future demands without compromising user experience or business continuity.

	Existing	Projected
EC2 CPU	16	32
EC2 Memory	64 GB	128 GB
RDS CPU	16	32
RDS Memory	64 GB	128 GB
RDS Type	SQL Web Edition	SQL Standard
RDS Storage	12 TB	24 TB
IOPS	20,000	40,000

Table 2: The observability data collected by eG Enterprise allowed a new specification to be defined for the scaled system.

04 Application Upgrade and Performance Issues

The migration to the upgraded configuration — involving the move from SQL Server Web Edition to SQL Server Standard Edition on Amazon RDS, along with the increased CPU and memory capacity for both the EC2 web tier and the RDS database tier — was executed smoothly. Deployment automation via Terraform and configuration management through AWS Systems Manager ensured the transition occurred with minimal downtime or service disruption.



Post-migration, the application performed exactly as expected. Response times were low, CPU utilization remained within acceptable limits, and both application and database uptime were consistently high. The ERP environment appeared stable, and the IT team was optimistic that the system was well-prepared to handle the anticipated growth in workload.

However, once the new stores were onboarded and the number of active users surged within just a week, the situation changed dramatically. Almost immediately, users began reporting slowness across key workflows such as inventory updates, billing transactions, and daily reconciliation processes. Before user complaints even reached the helpdesk, eG Enterprise had already detected and flagged early indicators of performance degradation.

In keeping with the complexity of most multi-tier cloud applications, the problem did not originate from a single component. Instead, multiple tiers began exhibiting symptoms simultaneously — from the application servers to the database tier. The alert dashboard of eG Enterprise showed multiple warnings and critical alerts spanning different tiers.

Database Latency Spike

The first clear indicator of distress came from eG Enterprise's application performance monitoring (APM) module, which captured real-time SQL insert times from the Java application to the Microsoft SQL backend. As shown in Figure 2, the average insert time — which had previously been less than 1 millisecond — spiked sharply, reaching values exceeding 150 milliseconds.

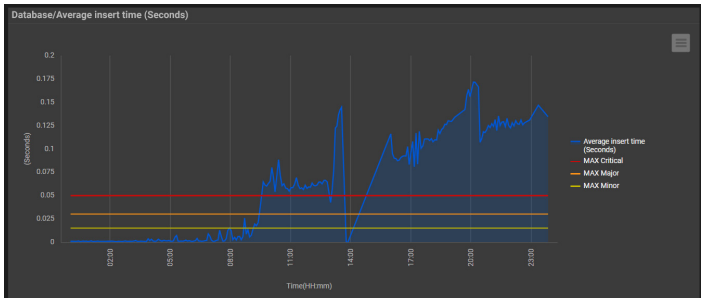
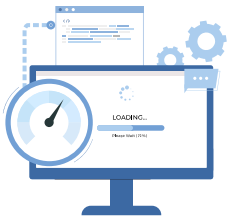


Figure 2: Average insert times to the database tables showed a dramatic increase, as measured by eG Enterprise.

This metric provided a strong early signal that database write operations were taking significantly longer than expected, creating transaction bottlenecks and cascading delays in user-facing application workflows. The average insert time rose progressively throughout the day as user load increased, crossing critical thresholds defined (the red line in Figure 2).

eG Enterprise's performance graphs helped the IT team visualize how rapidly the degradation occurred and when. However, as subsequent analysis would show, the database was only part of a broader chain of interrelated issues impacting the overall application stack.

Back-Pressure on the Application Layer



As the database tier struggled to handle the increased number of transactions, back-pressure began building up at the application tier. The delays in database insert operations caused requests from users to queue up within the Java web application, which was hosted on Apache Tomcat.

The result was a sharp increase in the number of active JVM threads, as each thread waited for database transactions to complete before it could process subsequent requests. eG Enterprise's JVM monitoring clearly revealed this behaviour.

Figure 3: The number of threads on the JVM hosted on Apache Tomcat also showed a marked increase in the eG Enterprise console.

As shown in Figure 3, the number of active threads in the JVM steadily climbed over the course of the day, peaking at approximately 1,500 threads — the maximum configured for Tomcat's connection pool.

Once this upper threshold was reached, new incoming user requests could no longer be accepted by the application. Clients attempting to log in or perform transactions began experiencing timeouts, "Service Unavailable" errors, or extremely slow responses.

At this stage, performance degradation cascaded across multiple layers:

- The application tier became saturated with waiting threads.
- The database continued to experience elevated write latencies.
- The load balancer began queuing requests for unresponsive instances.
- End-user experience deteriorated rapidly, with widespread reports of timeouts and page load failures.

In essence, the bottleneck in one tier propagated through the entire application stack, bringing the system close to a complete stall. eG Enterprise's correlated multi-tier monitoring allowed the IT team to trace this chain reaction — from slow SQL insert times, to thread pool exhaustion, and finally to application-level unavailability — in a matter of minutes.

Thread-level analysis from eG Enterprise pinpointed the exact location in the code where the execution was blocked. The stack trace in Figure 4 shows that the application is stalled during socket read operations to the SQL Server backend.

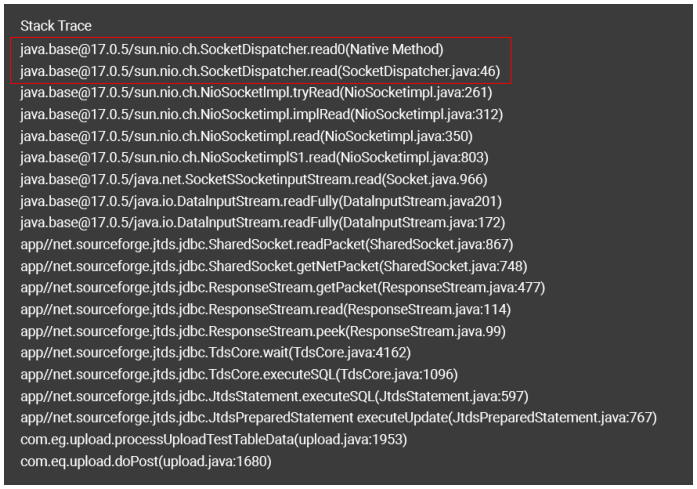


Figure 4: Stack trace data showed many threads suspended on the method sun.nio.ch.SocketDispatcher.read().

Each thread was suspended at the *sun.nio.ch.SocketDispatcher.read()* method — an indication that the application was waiting for data to be returned from the database. This consistent behaviour across hundreds of active threads suggested a systemic delay in responses from the RDS instance, rather than a problem in the application layers. A sudden increase in used connections in the connection pool from the application to the database tier confirmed a database service issue (see Figure 5).

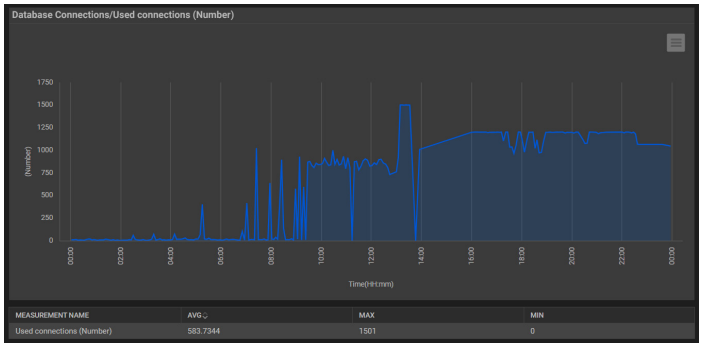


Figure 5: The sudden increase in used connections pointed to a database service issue.

An Unexpected Finding About the Database Tier

While the earlier thread analysis strongly pointed toward a database bottleneck, a deeper look at Amazon CloudWatch metrics, correlated through eG Enterprise, uncovered an unexpected twist.

The IOPS utilization of the Amazon RDS instance — a key indicator of database read/write workload intensity — was surprisingly low. As shown in Figure 6, throughout the observation period, IOPS utilization averaged just around 8%, with a maximum of 28% — far below the thresholds that would suggest the storage subsystem was being saturated.

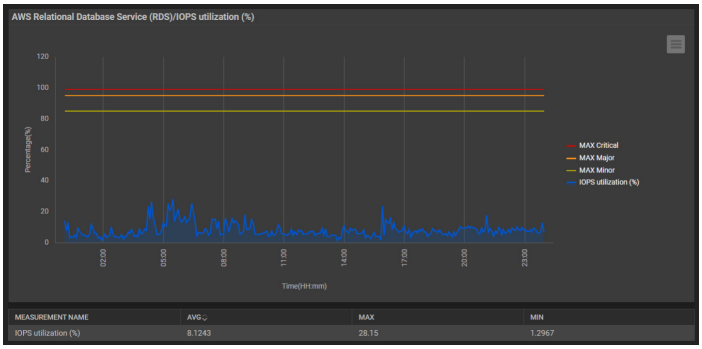


Figure 6: Curiously the IOPs utilisation of the RDS instance did not exhibit signs of issues.

This observation contradicted the initial assumption that the RDS instance was under stress due to high disk I/O demands. Typically, when a database becomes a bottleneck due to excessive transaction load, one would expect to see IOPS utilization approaching 100%, indicating that the underlying storage layer is overwhelmed. However, in this case, the storage subsystem appeared to be underutilized.

It's NOT a Resource Bottleneck in the Database Service

As the team investigated further, another unexpected pattern emerged. Disk read latency, as reported by Amazon CloudWatch and analyzed through eG Enterprise, remained extremely low throughout the period of user complaints. This ruled out any possibility of storage slowness or queue build-up at the disk subsystem level. In parallel, when the team examined the SQL Engine's CPU utilization (see Figure 7), the results were equally surprising. Even during peak hours, when users were experiencing long delays and application slowdowns, the average CPU utilization of the SQL Server process was only around 9%, peaking at just 17%. For a system expected to be handling three times the previous transaction load, such low CPU and IOPS activity indicated that the RDS instance was not truly processing a higher workload — rather, it was waiting on something.

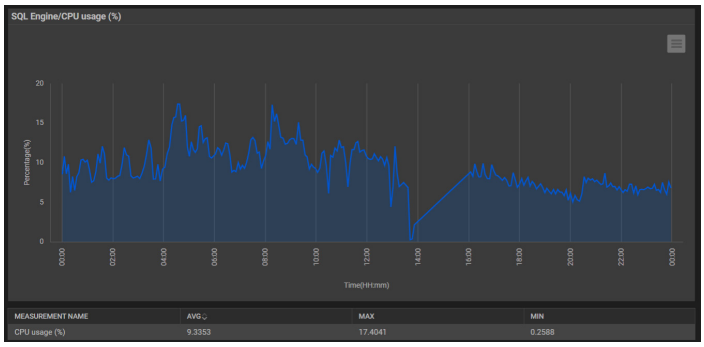


Figure 7: Average CPU utilization of the SQL Server process was only around 9%, suggesting that the RDS instance was not truly processing a higher workload.

Confirming the Nature of the Bottleneck

To further isolate the root cause, the IT team examined SQL user process activity on the RDS instance using eG Enterprise's database monitoring. The expectation was to see a surge in active (running) processes during peak load periods if the SQL Server engine was indeed processing a larger number of concurrent queries.

However, as Figure 8 shows, this was not the case. Most of the SQL user connections were in a "sleeping" state, meaning they were idle and waiting for commands from the application. A small fraction were suspended, indicating that the queries had been issued but were waiting for locks or resources to become available. The number of actively running processes remained extremely low — averaging just two concurrent processes throughout the day.

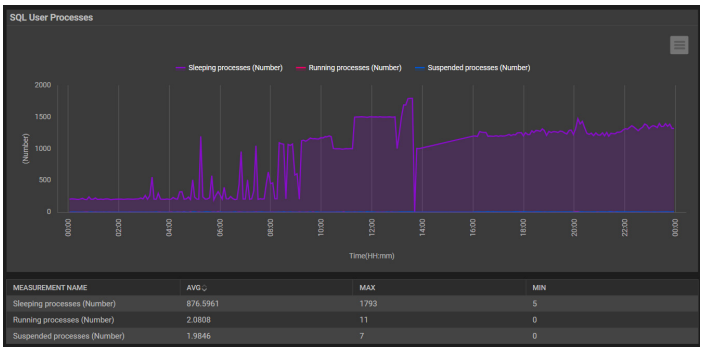


Figure 8: eG Enterprise revealed that most SQL user connections were in a "sleeping" state, meaning they were idle and waiting for commands from the application.

Network Latency Revealed – The Real Culprit Emerges

While the internal metrics of the AWS RDS instance showed no signs of stress, synthetic monitoring checks performed by eG Enterprise provided the breakthrough insight that helped isolate the true cause of the issue. These synthetic checks simulated real-world database transactions by periodically initiating connections from the EC2 web server to the RDS SQL Server instance, executing lightweight queries, and measuring the connection and query response times. The results were eye-opening.

As illustrated in Figure 9 below, there was a sharp increase in the connection time to the SQL service, while the query execution time was not significantly high.

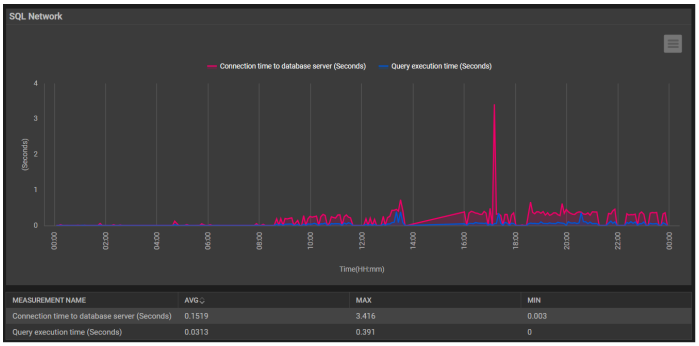


Figure 9: Analyzing the change in connection time to the database service and query execution time. Most of the latency increase was due to increased connection time to the database service.

In Figure 9, the connection time to the database (represented in pink) spiked to over 3 seconds during peak periods, even though the query execution time (blue line) stayed below 0.4 seconds. This clearly indicated that the delay was occurring before the query even began execution — i.e., during the establishment of a TCP connection between the application and the database. In other words, the database itself was ready and responsive, but network-level latency or connection handshake delays between the EC2 and RDS instances were impeding performance.

Verifying the Network Latency

To validate the findings from eG Enterprise's synthetic monitoring, the IT team decided to conduct independent tests using low-level network utilities. By manually executing the tcping command from the EC2 instance to the RDS SQL Server endpoint, they were able to measure the round-trip time for establishing TCP connections.

```
Probing 10.0.0.252:1433/tcp - Port is open - time=66.617ms
Probing 10.0.0.252:1433/tcp - Port is open - time-67.539ms
Probing 10.0.0.252:1433/tcp - Port is open - time=89.022ms
Probing 10.0.0.252:1433/tcp - Port is open -time=58.646ms

Ping statistics for 10.0.0.252:1433
    4 probes sent.
    4 successful, e failed. (0.00% fail)
Approximate trip times in milli-seconds:
    Minimum - 58.646ms, Maximum - 89.022ms, Average - 70.456ms
```

Figure 10: Results of 'tcping' checks to the database service. While normal latency was less than 1 msec, here the latency value is several fold above the norm.

The results confirmed what eG Enterprise had already indicated — the latency between the EC2 application server and the RDS database instance was significantly higher than normal. In some cases, connection times spiked several times above the baseline, even though both systems were hosted within the same AWS region and Virtual Private Cloud (VPC).

This direct network testing eliminated any doubts about application or database misconfiguration. The elevated latency was a network-layer issue affecting TCP handshakes and connection establishment between the two instances.

Cloud Provider: "It's Not Us"

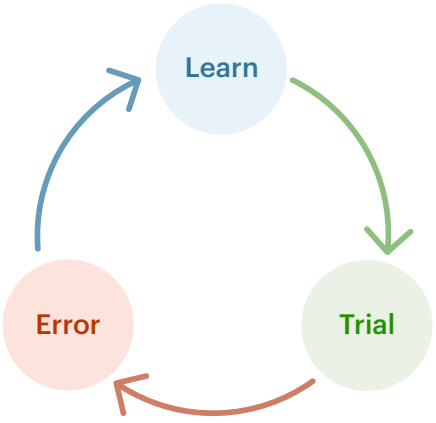
Despite the clear technical evidence presented — including time-correlated graphs from eG Enterprise, synthetic test results, and tcping latency data — the cloud service provider declined to acknowledge the problem. The official response to the customer's escalation was a familiar refrain: "Everything is fine from our end."

This lack of support left the IT operations team with limited options. They had to rely on their own observability data and correlation analysis to prove the existence of a problem that was transient, region-specific, and outside their direct control.

The situation highlights a crucial reality of cloud operations: even though the infrastructure is managed by the provider, the responsibility for ensuring service performance and availability still lies with the customer.

05 Identifying the Performance Bottleneck

Left with limited options and without support from the cloud provider, the application operations team decided to focus on what they could control — the database connectivity layer of the application itself. Since all evidence pointed toward connection-level latency between the EC2 instance and the RDS database, the team began experimenting with SQL driver optimizations to minimize potential transport and protocol overheads.



Database Driver Configuration Tuning

A variety of configuration parameters and driver-level options were systematically tested. The key optimizations attempted included:

- **Disabling TcpNoDelay** to allow packet coalescence and reduce the number of small network transmissions.
- **Disabling Unicode string conversion**, minimizing the need for encoding transformations and reducing payload size.
- **Increasing the TDS packet size** (the SQL Server wire protocol unit) to enable more data to be transmitted per round-trip and reduce network chattiness.

Additionally, the team experimented with different Java Database Connectivity (JDBC) drivers — including both JTDS and Microsoft's official SQL Server JDBC drivers — to see if any implementation handled connection establishment or buffering more efficiently.

Despite these exhaustive trials, none of the driver-level optimizations produced any measurable improvement in performance. The average connection latency remained elevated, and user-facing application response times continued to degrade during load surges. This conclusively demonstrated that the issue was not within the control of the application stack or driver configuration, but was instead an infrastructure-level network latency problem between the compute and database layers.

These findings reinforced an important operational lesson: Without complete observability across application, network, and cloud service layers, IT teams can easily waste valuable time optimizing the wrong components.

With end users growing increasingly frustrated and executive attention now drawn to the issue, the application operations team explored additional options to relieve the performance bottleneck. Since all evidence pointed to connection latency between the EC2 web tier and the RDS SQL Server instance, the focus turned to connection management within the application.

Connection Pool Adjustments

The team decided to increase the initial number of connections in the database connection pool used by the application. The rationale was that maintaining a larger pool of pre-established connections could minimize the need for frequent TCP handshakes, thus mitigating connection latency effects.

However, this tuning yielded only marginal improvement. While it helped sustain throughput for short periods, it did not resolve the core issue — the persistent and unpredictable delays in establishing new database connections.

Database Memory Reconfiguration

During the investigation, it was also discovered that the RDS SQL Server instance had been configured to consume over 90% of the system's available memory for database buffer management. While this configuration maximized caching efficiency under normal conditions, it left very little memory available for the operating system and TCP stack.

This imbalance likely contributed to sluggish network connection handling on the RDS side — especially under heavy concurrent connection churn. To address this, the database team reconfigured the SQL Server memory settings to limit database memory usage to 75% of total

system memory, thereby freeing resources for the operating system and network subsystems.

Unfortunately, neither of these changes had a significant impact on end-user experience. Connection times remained inconsistent, transaction latency persisted, and the backlog of user complaints continued to grow.

By this point, the issue had escalated to senior management, prompting daily status reviews and executive oversight. It was clear that surface-level optimizations were insufficient; a deeper, systemic root cause analysis was required.

Widespread Impact Across Application Workloads

The performance degradation was not confined to interactive user sessions alone. As the latency issue persisted, its impact began to manifest across background and scheduled workloads as well.

Many hourly batch jobs — which were critical for synchronization, data analysis, and reporting — started exhibiting severe slowdowns. These jobs were designed to execute sequentially, performing database inserts and updates in bulk. Under normal circumstances, each batch cycle completed within 20 to 30 minutes. However, with the onset of the latency problem, batch job durations ballooned to nearly five hours. Figure 11 below illustrates how the *time taken for batch processing* metric steadily increased throughout the day, mirroring the same pattern observed for interactive user operations.

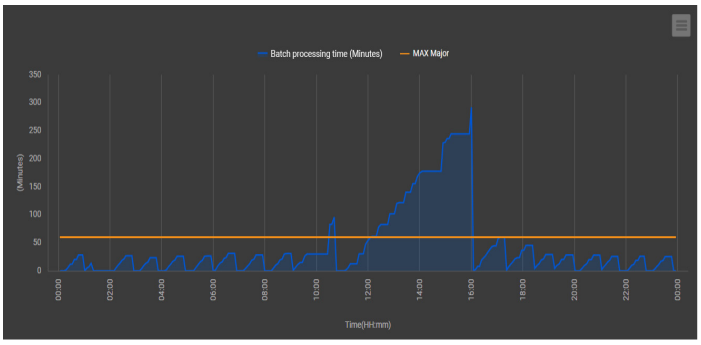


Figure 11: Batch processing tasks that took less than 30 mins were taking 5 hours at peak load.

Impact beyond Slowness

This cascading slowdown had far-reaching operational implications:

- **Data processing pipelines** began overlapping with subsequent job schedules, leading to concurrency conflicts.
- **Dependent analytics tasks** were delayed, impacting downstream reporting systems.
- **System administrators** faced increasing alerts as SLAs for job completion were repeatedly breached.

The Breakthrough – Identifying the Network Transport Issue

As the troubleshooting efforts intensified, a critical clue finally surfaced from the network layer monitoring of eG Enterprise. While the focus had so far been on application, database, and driver-level parameters, eG Enterprise began generating alerts indicating a surge in TCP retransmissions from the EC2 application instance. Figure 12 below shows how the TCP retransmit ratio — which had historically been close to zero — began spiking sharply during periods of high workload. At peak load, TCP retransmissions exceeded 20%, and in some intervals, they climbed as high as 50% of total packets sent.

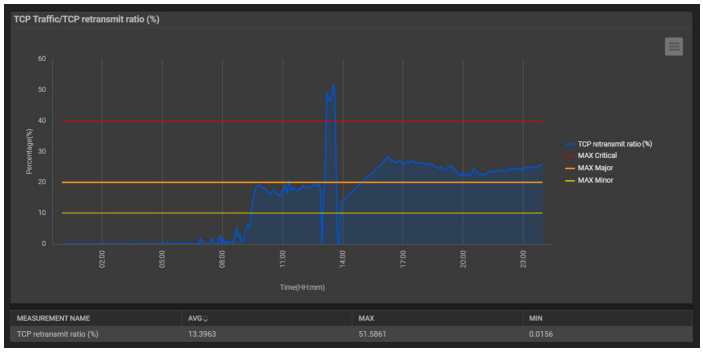


Figure 12: TCP retransmits increased with load.

TCP retransmissions occur when packets transmitted over the network are either lost or delayed, forcing the sender to retransmit them. A high retransmission ratio is a clear sign of packet loss, congestion, or unstable network links. In this case, the increase in retransmissions explained the slow connection establishment and sporadic timeouts observed between the EC2 and RDS instances.

Each retransmission delayed TCP handshake completion and degraded throughput, causing:

- **Longer connection setup times** to the SQL database
- **Application thread pile-up** waiting on blocked sockets
- **Exponential slowdown** in user transactions and batch job execution

At this stage, after exhaustive analysis across all tiers of the application stack, the evidence was overwhelming and conclusive — the SQL Server instance was not the problem, and the application logic itself was functioning correctly.

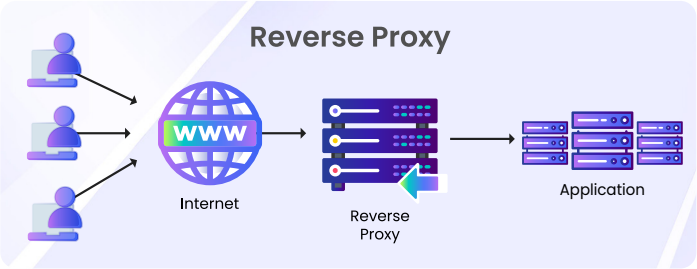
> **The consistent pattern of high TCP retransmissions, intermittent connection delays, and low database utilization all pointed to a single remaining suspect: network packet processing within the EC2 instance. The problem was not what the application or database were doing, but how the data was being transmitted between them.**

06 Resolving the Performance Bottleneck

A joint "Tiger Team" was put together including members from the application architecture, network, and database teams. Their mandate was to determine how to resolve the issue. They had to determine whether the configurations of the EC2 and RDS instances needed to be increased, if a different means of network routing was needed, etc. After days of deliberating and analysis, this team identified three changes needed in the application deployment.

Enabling Keep-Alives on the Application Server

One of the most impactful insights came from analyzing how client requests were reaching the application tier. The Application Load Balancer (ALB) deployed in front of the web application acted as a reverse proxy, managing and multiplexing all incoming connections from thousands of geographically distributed store clients. This meant that, contrary to initial assumptions, the application servers were not directly communicating with thousands of end-user clients. Instead, they were maintaining persistent connections only with the ALB.



This architectural realization opened the door to a simple yet powerful optimization: enabling HTTP Keep-Alives on the application server. Previously, Keep-Alives had been disabled out of caution — to avoid a scenario where thousands of idle client connections from remote stores would consume the server's socket and memory resources. However, with the ALB now acting as the single upstream peer, the risk of connection exhaustion was eliminated.

The team determined that by allowing persistent Keep-Alive connections between the ALB and the Tomcat application tier, the following would be reduced:

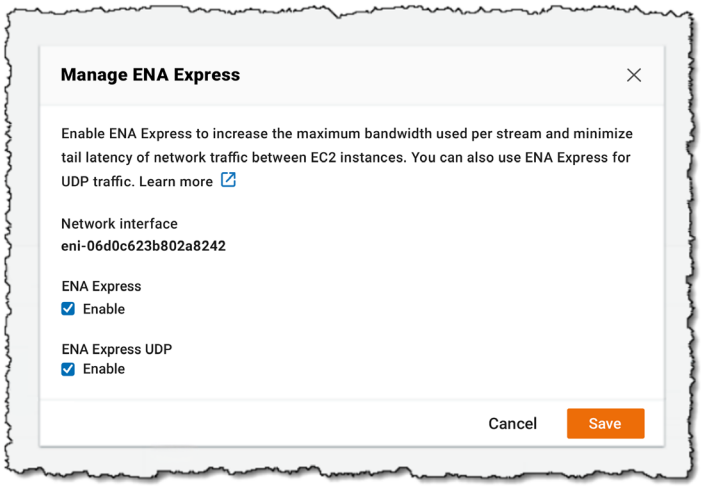
- The overhead of repeatedly establishing and tearing down TCP sessions.
- The frequency of short-lived TCP handshakes that contributed to retransmissions.
- Latency caused by connection setup delays for each new request.

The change required minimal configuration — a few parameter updates on the Tomcat connector settings — but it delivered an immediate improvement in stability. Connection reuse meant far fewer transient network issues, reduced packet retransmissions, and lower connection setup latency to the database. This seemingly small configuration change marked the first major breakthrough in restoring performance and demonstrated the value of cross-team collaboration and architectural visibility.

Enabling ENA Express for Enhanced Network Performance

The second decisive step involved improving the network transport performance of the EC2 instance itself. Given that high TCP retransmissions had been identified as the root symptom, it was clear that the underlying network interface needed to handle larger volumes of concurrent traffic more efficiently. To achieve this, the infrastructure team decided to migrate the instance from an *m5.8xlarge* type to a

newer-generation *m6in.8xlarge* instance. This change was strategic — it did not alter CPU, memory, or storage capacity, but it introduced support for AWS's latest network acceleration technology: Elastic Network Adapter (ENA) Express.



ENA Express is designed to enhance packet processing performance for latency-sensitive applications. It leverages AWS Scalable Reliable Datagram (SRD) technology to improve throughput, reduce tail latency, and deliver more consistent network performance within a region. ENA Express provides:

- Lower network jitter and packet loss by optimizing traffic between EC2 instances and AWS-managed services like RDS.
- Higher bandwidth utilization and more efficient packet reordering at high concurrency levels.
- Reduced TCP retransmissions, especially under bursty transaction loads.

Applying Windows OS and Network Stack Optimizations

The third set of changes focused on the Windows operating system's TCP/IP and network configuration on the EC2 instance. Even though ENA Express improves network throughput, one needs to ensure that the underlying TCP stack and adapter parameters are tuned to handle the heavy transactional load generated by thousands of client connections.

- **Operating System-Level Enhancements:** A number of Windows TCP/IP best practice settings were applied to optimize socket management, connection reuse, and buffer sizing. These adjustments included lowering TCP connection timeouts, expanding the range of available ephemeral ports, and increasing the number of free transmission control blocks (TCBs). These changes aim to:
  - Prevent socket exhaustion under burst traffic conditions.
  - Enable faster reuse of closed connections.
  - Maintain efficient connection handling even under sustained high concurrency.
- **Network Adapter Buffer Optimization:** The receive and transmit buffer sizes on the EC2 network adapters were increased substantially. This allowed the system to handle higher packet rates with fewer dropped frames or retransmits, ensuring smoother throughput when large data bursts occurred between the application and RDS tiers.
- **TCP Stack Tuning:** Finally, several advanced TCP parameters were fine-tuned to improve reliability and compatibility in the virtualized AWS environment. This included disabling unnecessary features such as Receive Segment Coalescing (RSC) and Explicit Congestion Notification (ECN), while enabling resiliency features that improve recovery from transient network delays. Collectively, these refinements enhanced how Windows handled acknowledgments, retransmissions, and packet pacing, leading to far more predictable network performance.

The Results

After implementing all three sets of optimizations, the transformation was dramatic. The application's performance not only returned to normal but improved well beyond its pre-expansion levels.

- TCP retransmissions dropped to less than 1%, even during peak transaction periods.

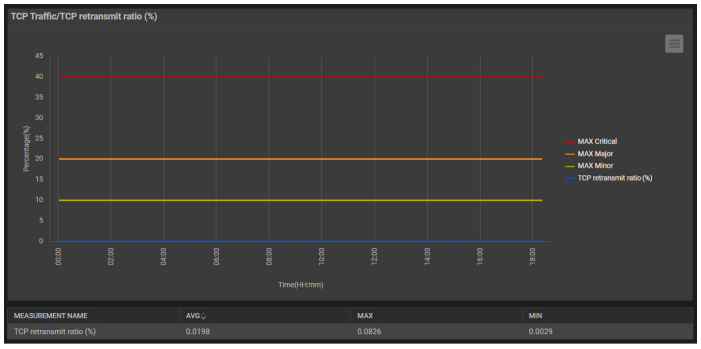


Figure 13: TCP retransmits remained near zero throughout.

 |

- Latency to the SQL database was consistently below 1 second, ensuring smooth real-time interactions.

Figure 14: Total time to connect and query the database dropped significantly below 1 msec.

- Average database insert times, which had spiked to over 150 milliseconds during the crisis, stabilized to well below 1 millisecond, indicating efficient data flow between the application and database.

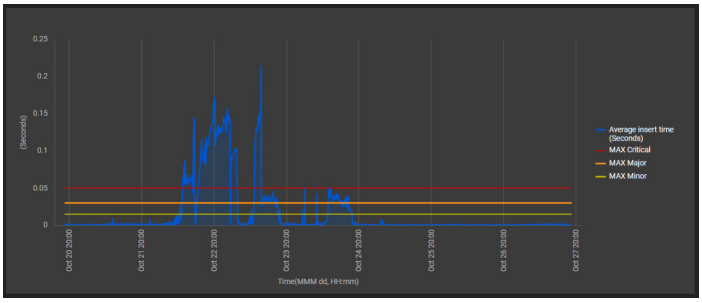


Figure 15: Database insert times which had reached 150 msec during the problematic period were now less than 1 msec even with peak load.

- The application scaled effortlessly to handle over 10,000 stores — more than three times the original workload — without any instability or timeout issues.
- AWS RDS IOPS utilization increased, reflecting healthy database activity, yet it remained comfortably below the allocated capacity, confirming that the storage subsystem was no longer the bottleneck.
- Batch processing times dropped back to their expected duration of 20–30 minutes, fully within service-level expectations.

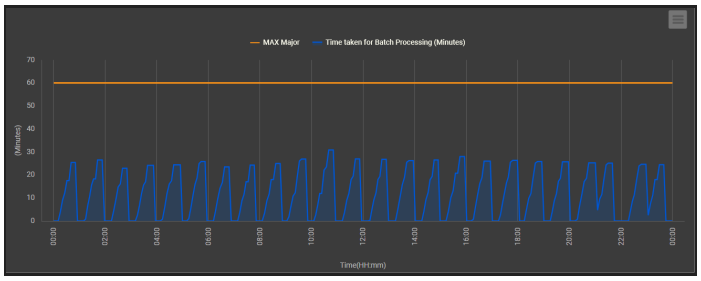


Figure 16: Batch processing time remained around 30 mins throughout the day, highlighting the capability of the tuned system to handle peak load well.

07 Eight Real-World Lessons Learned

Here are eight lessons we learned from assisting with the troubleshooting of the ERP application's performance in the cloud:

1. **More Capacity ≠ More Performance:** Scaling compute and memory alone cannot guarantee responsiveness. The real determinants of performance often lie in network throughput, TCP tuning, connection management, and storage latency. Without end-to-end visibility, additional capacity may only mask the symptoms, not fix the root cause.

2. **Every Layer Matters:** High-performance cloud applications demand meticulous attention to detail — instance family, NIC type, driver version, storage class, SQL memory allocation, and database durability settings all influence throughput. The smallest misconfiguration can create ripple effects across tiers.

3. **Network Efficiency Defines Application Experience:** TCP retransmissions, buffer sizing, and NIC queue configuration can make or break performance. In this case, ENA Express and optimized adapter settings dramatically reduced latency. Network optimization is as essential as database indexing or JVM tuning.

4. **AWS RDS Simplifies Operations but Limits Visibility:** Managed services like AWS RDS abstract infrastructure but also constrain troubleshooting. Metrics such as IOPS or CPU alone cannot reveal transient connection-layer issues. For mission-critical workloads, deploying SQL Server on EC2 rather than as a managed service provides the transparency required for deep diagnosis.

5. **Observability Is the Accelerator for Resolution:** eG Enterprise delivered unified, correlated visibility — from Java thread states and connection pools to RDS IOPS, TCP retransmits, and CloudWatch data. This single-pane correlation pinpointed that the issue wasn't SQL or the application but the EC2 network stack — saving days of blind debugging.

6. **Cross-Functional Collaboration Is Not Negotiable:** Performance crises transcend silos. The breakthrough came when application, network, and

database teams worked together, using common evidence from eG Enterprise dashboards. Multi-connector design and separate thread pools web access, database access, etc. made it easier to isolate the failing path.

7. **Cloud Provider Support Alone Is Not Enough:** Even with premium support contracts, the cloud provider's scope ends at the service boundary. "Everything is fine from our end" is a common reponse. Enterprises must build self-reliant diagnostic capabilities and retain data that clearly demonstrates cross-service issues.

8. **Structured Troubleshooting Delivers Sustainable Results:** Root-cause analysis is iterative — measure, hypothesize, test, and validate. With correlated metrics and historical baselines in eG Enterprise, teams could quantify each change's impact (e.g., keep-alives, ENA Express, TCP tuning) and confirm improvement objectively.

The eG Enterprise Advantage

eG Enterprise played a pivotal role throughout this engagement. It provided:

- **Unified observability across cloud, OS, application, and database tiers,** eliminating guesswork.
- **Historical baselining and trend analytics,** enabling the team to prove that performance degradation aligned precisely with increased network retransmissions.
- **Transaction-level visibility,** showing that application threads were waiting on SQL responses — not JVM or business-logic bottlenecks.
- **Synthetic monitoring and CloudWatch integration**, confirming that the problem lay between EC2 and RDS network paths.
- **Real-time validation of fixes**, instantly reflecting the impact of keep-alive enablement, ENA Express migration, and TCP stack optimization.

With these capabilities, eG Enterprise transformed a multi-week cross-team firefight into a data-driven, one-week resolution — restoring stability, scalability, and confidence in the cloud deployment.