IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p-ISSN: 2278-8727, Volume 28, Issue 1, Ser. 2 (Jan. – Feb. 2026), PP 72-78
www.iosrjournals.org

A Comparative Study Of Centralized And Distributed Payment Gateway Architectures

Vimal Teja Manne

Abstract
Payment gateways sit at the choke point of checkout, they translate a customer's "Pay" click into a sequence of validations, authorisation calls, and recorded outcomes that merchants and banking networks can trust. When volume is low, architecture barely matters, a single well-built gateway can look flawless. As throughput climbs, the design starts to leak into everyday reality: reliability, scalability, how often on-call gets paged, and whether a slow dependency becomes a minor annoyance or a full outage. Centralized gateways stay popular because they are simple to build and operate, one main request path, one policy surface, one transaction store, and they can deliver steady performance at moderate load, but that same central path concentrates risk and tends to hit hard scaling limits once queues and shared state become hot. Distributed gateways spread processing across multiple nodes so capacity can grow horizontally and failures can degrade service instead of stopping it, but they carry coordination overhead, consistency pressure around retries and idempotency, and a larger opera- tional surface that demands stronger observability. This paper compares the two architectures using practical performance and reliability lenses, latency under rising load, fault tolerance during component failures, scalability behavior, and operational complexity, using simulated transaction workloads to surface trade-offs across different operating conditions without treating either model as a universal answer.

Index Terms: *Payment gateway, centralized architecture, dis- tributed systems, electronic payments, transaction processing, system performance*

Date of Submission: 07-02-2026 Date of Acceptance: 17-02-2026

I. Introduction

Electronic payment systems have become routine infrastruc- ture for day-to-day commerce, carrying transactions across retail platforms, service providers, and financial institutions [1]. In most online transactions, the payment gateway acts as the coordination point between customers, merchants, and banking networks. When this layer slows down or becomes flaky, users feel it immediately, checkout drop-off climbs, and merchants take the blame even if the root cause sits elsewhere [2].

Many early gateway deployments were built around a cen- tralized design. A single gateway server, or a tightly coupled cluster behaving like one, handled transaction routing, valida- tion, and the core data path [3]. For small to mid-sized systems this was a practical choice because it reduced integration surface area and kept operations simple. The trade-off shows up as load grows. Centralized gateways run into capacity bottlenecks faster than teams expect, and they concentrate failure risk, so a bad deploy, a database choke point, or a regional outage can turn into a full payment stoppage [4].

Distributed payment gateway architectures grew out of the same pain points operators see in the field, saturation at peak load, brittle failover paths, and the reality that a single "gateway brain" becomes an outage multiplier. By spreading transaction intake and processing across multiple nodes, a distributed gateway can keep accepting payments even when a subset of components misbehaves, and it can expand throughput by adding capacity horizontally instead of constantly upgrading one central box [5]. That resilience is not free. Once transaction handling is split across nodes, the gateway has to coordinate decisions, keep shared state aligned tightly enough to prevent double charges or missed authorizations, and preserve observability that still reads like a single story when one checkout request fans out across several services and hops [6].

Centralized and distributed designs both appear in transac- tion processing work, but payment gateways have quirks that make a focused comparison worthwhile. They sit at the bound- ary between merchants and financial networks, correctness is non-negotiable, and success is usually measured in failure behavior and tail latency rather than average throughput. Using simulated workload scenarios, this study compares both approaches by looking at reliability under component failures, latency behavior as load rises, and the operational trade-offs teams take on when they commit to one architecture.

II. Related Work

Prior work covers online payment systems from several angles, often with strong practitioner and community in- fluence. Laudon and Traver frame the gateway as a cen- tral e-commerce component and make a blunt but useful point: security and dependability matter most at the exact choke point where checkout traffic converges [1]. Greenstein and Feinman discuss electronic commerce through a risk- management lens, and those concerns frequently shape how real organizations structure payment flows, choose providers, and enforce controls in production [3]. A large share of the payments literature still clusters around fraud prevention, encryption, authentication, and related security mechanisms [4]. That work is necessary, but it often treats the gateway as a single logical box and moves on, leaving the ugly engineering reality in the margins: traffic spikes that push queues into saturation, mid-authorization failures that trigger retry storms, idempotency edge cases that show up only under partial outages, and rollout constraints where updates must land without pausing transaction intake. As a result, fault tolerance and scalability are not always studied in a way that resembles how gateways behave when they are stressed. In parallel, distributed systems research offers deep coverage of replication, failure models, and the trade-offs that appear once state and execution are spread across nodes [5], and service-oriented work highlights the operational cost of decomposition, coordination, and cross-service visibility [6]. What is still thin is a payment-gateway-specific treatment that carries those concepts into a domain where correctness is non-negotiable, tail latency is the real KPI, and failures are both expensive and immediately visible to merchants and customers. Direct, payment-gateway-specific comparisons of centralized and distributed architectures remain surprisingly thin. Plenty of work discusses transaction systems in broad terms or approaches payments mainly through a security lens, but far fewer studies put the two architectural choices side by side and judge them against gateway realities such as authorization round trips, retry-driven idempotency pressure, and the unpleasant behavior that shows up in the tail of the latency curve. This study builds on the existing foundations but narrows the focus to what gateway teams actually live with, offering a practical comparison that treats the gateway as the system under test rather than a footnote inside a larger platform.

There is strong prior work on performance, scaling, and reliability in large-scale transaction processing. Classic trans- action processing concepts and consistency models are well- covered in foundational literature [7], [8]. Cloud computing and service-oriented deployment styles have been studied as ways to improve elasticity and availability, especially when systems move from a single stack to distributed components [9], [10]. Network behavior matters too, because a gateway's critical path often includes multiple external hops, and latency variability on the Internet can dominate the end-to-end re- sponse time even when the application code is efficient [11]. These threads provide the conceptual toolbox, but they do not automatically settle the centralized vs distributed decision for payment gateways.

III. Centralized Payment Gateway Architecture

In a centralized payment gateway architecture, transaction requests funnel through a single processing unit that owns the core workflow, input validation, authentication steps, encryp- tion or tokenization, and communication with upstream bank- ing networks, and transaction records are typically persisted in one central database, which keeps auditing, reconciliation, and operational monitoring relatively straightforward [3]. The simplicity is real: one request path, one control plane, one place to enforce policy and reason about behavior, and that often translates into stable, predictable latency at moderate load because there is less cross-service coordination and fewer opportunities for partial failure. The flip side is that centraliza- tion puts both the gateway process and its database directly on the critical path for every checkout, so saturation or instability shows up immediately as a platform-wide symptom, not a localized incident. Risk is concentrated too, if the gateway host fails, or the database becomes unavailable or I/O-bound, the payment flow can stall completely because there is no alternate processing path. Scaling is bounded by the vertical limits of the central stack (CPU, memory, database write throughput, network bandwidth), and once traffic pushes beyond those limits performance often falls off a cliff rather than tapering, with queues building, timeouts rising, and retries turning a capacity problem into a self-amplifying retry storm [4].

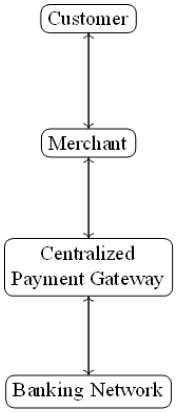

Figure 1. Centralized Payment Gateway Architecture

IV. Distributed Payment Gateway Architecture

A distributed payment gateway splits transaction handling across a fleet of processing nodes instead of pushing every request through a single gateway instance. Requests are usu- ally fronted by a load balancer and, in more mature setups, a routing layer that can make smarter choices than round- robin, for example draining a degraded node, pinning certain traffic classes, or keeping bank-specific integrations away from overloaded workers. The moment you fan out processing, transaction state can't sit comfortably inside one process anymore. Idempotency keys, authorization state, settlement records, and any ledger-style updates have to be shared in a way that remains correct under retries, partial timeouts, and node churn, which is why distributed gateways lean heavily on replication or synchronization mechanisms (and the failure trade-offs that come with them) rather than a single authoritative in-memory view [5].

The day-to-day win is resilience. If a node drops out, the gateway usually keeps taking traffic, you lose some through- put, but you do not lose the whole checkout path, so incidents show up as reduced capacity and higher latency rather than a hard outage. Scaling is also less awkward than in a single- instance design, since you can push throughput by adding more workers and spreading load across them, which maps cleanly onto cloud elasticity and the reality that payment traffic arrives in bursts, not smooth lines [9].

The trade-offs are not subtle. Once transaction handling is split across nodes, coordination becomes a first-class problem, and consistency stops being a background detail you can hand-wave away, especially under retries, timeouts, and partial failures where the same authorization can be attempted twice unless the idempotency path is airtight. Observability gets harder in exactly the way that hurts during incidents: one payment attempt may bounce through a router, a worker, a message queue, an idempotency store, a risk service, and a bank connector, and without end-to-end tracing plus disci- plined correlation IDs, you end up with logs that are techni- cally "there" but useless as a narrative. That complexity shows up everywhere, in design, in deployments, in runbooks, and in the ongoing cost of keeping the system understandable, even though the customer only experiences a single "Pay Now" click [6].

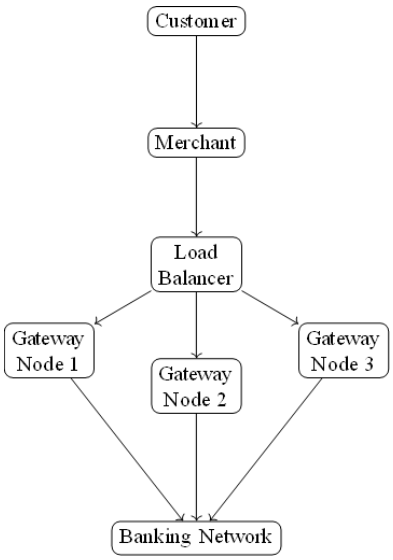

Figure 2. Distributed Payment Gateway Architecture

V. System Model And Assumptions

The system studied is a generic payment gateway with four roles: customers initiating payments, merchants requesting authorization, a gateway layer that mediates the workflow, and external banking networks that ultimately approve or decline transactions. The evaluation uses simulated workloads only, with no real financial records, cardholder data, or personally identifiable information.

Each transaction is modeled as a request–response flow with three stages that mirror common gateway behavior: authentication (verifying the caller or session), authorization (requesting approval from the banking side), and confirmation (recording the outcome and returning the final status). Network delay is held uniform to keep the comparison focused on ar- chitectural effects rather than Internet path variability. Failures are injected at the server or node level (crash, unavailability, or forced downtime) instead of modeling router, DNS, or upstream network faults, so the results reflect gateway-side behavior more than infrastructure edge cases.

For the centralized case, the gateway is modeled as a single logical processing unit backed by one centralized transaction database. For the distributed case, multiple gateway nodes run concurrently behind a load distribution layer, with shared state maintained through synchronization or replication mechanisms rather than being owned by any single node. The comparison is intentionally kept at the architecture level, focusing on the behavior implied by transaction processing and consis- tency constraints, not on performance tuning tricks or vendor- specific optimizations that can mask structural differences in the short term [7], [8].

VI. Experimental Setup And Evaluation Approach

A simulated test environment was built to compare cen- tralized and distributed payment gateway architectures un- der conditions that resemble everyday transaction processing. Rather than recreating any specific vendor gateway or payment processor, the setup is intentionally generic so the results reflect architectural behavior, what changes purely because the design is centralized or distributed, as load rises and components fail.

Workloads were generated as synthetic transaction requests with two broad patterns: a steady baseline rate and a higher- rate phase used to push the system into stress conditions. In the centralized setup, a single gateway instance handled all incoming requests and persisted outcomes to one transaction store. In the distributed setup, multiple gateway nodes ran in parallel behind a load balancer that routed requests across healthy nodes using basic availability checks. In both cases, every request followed the same simplified pipeline, authen- tication, authorization, then confirmation, so any differences in results come from the gateway architecture and its state- handling constraints rather than from differences in business logic.

The evaluation tracks the metrics that tend to reveal prob- lems early in real gateways: transaction latency (and how it drifts as load increases), throughput trends, and observed availability at each workload level. Fault tolerance is exercised by disabling gateway nodes during execution and watching whether processing continues, how quickly capacity collapses or degrades, and whether the system stabilizes once the failed node is removed or brought back. Results are read compar- atively rather than as absolute performance claims, since the intent is to expose architectural trade-offs without leaning on real payment data or vendor-tuned optimizations.

A. Evaluation Metrics

Latency is measured as end-to-end time, from the moment a transaction request enters the gateway to the moment the confirmation response is returned. Throughput is reported as the number of successfully completed transactions per unit time under a given offered load. Availability is evaluated operationally, by observing whether the system continues to process transactions when a gateway component is taken out of service, rather than against a theoretical uptime target. Scal- ability is assessed qualitatively by considering how behavior should change as additional processing nodes are added in the distributed configuration, using common cloud elasticity assumptions as the baseline [9], [10].

Table I
EVALUATION SETUP PARAMETERS (SIMULATED)

Parameter	Value / Description
Workload type	Synthetic transaction requests
Load levels	Moderate and increased request rates
Centralized model	Single gateway instance
Distributed model	Multiple gateway nodes + load balancer
Failure simulation	Disable one node during runtime
Observed metrics	Latency trend, availability behavior, scalability

VII. Comparative Analysis

The difference between centralized and distributed gateway designs shows up most in operational behavior, not archi- tecture diagrams. Both can implement the same checkout workflow, but they react very differently under pressure, bursty traffic, slow or jittery authorization responses, and partial failures that are routine in payment flows.

Transaction Latency

A centralized gateway typically has a short, direct critical path. There is minimal internal coordination because one instance owns decision-making and state transitions. At mod- erate traffic levels, that often yields stable response times with low variance, which is why centralized setups feel reassuring early on. As offered load rises, the same "one path" struc- ture becomes the bottleneck. Requests begin to queue, the database turns into a shared choke point, and latency climbs as backlogs form. Long before the system fully tips over, the user experience can still degrade sharply because tail latency usually expands first, the slowest few percent of transactions start timing out while averages remain deceptively calm in dashboards.

Distributed gateways usually pay a small latency tax be- cause the request path is no longer "one box, one database, done." Each transaction now includes routing to a healthy node, and in many designs it also includes coordination around shared state, for example checking an idempotency key, reserving an authorization record, or synchronizing a write so two nodes do not both believe they own the same transaction. That overhead is often modest per request, but it becomes visible when synchronization sits on the critical path or when shared stores get hot. Teams still accept it because it buys headroom and, more importantly, it avoids the classic failure mode where one overloaded gateway instance drags the entire checkout experience down with it.

Fault Tolerance and Reliability

Centralized gateways are structurally sensitive to failures in the gateway instance or its backing database. If either is unavailable, transaction processing usually stops until recovery completes. Redundancy and failover reduce the blast radius, but they do not erase it. Failover is disruptive by nature, and the act of switching can create its own transient problems, dropped connections, cold caches, leader election delays, or short windows where in-flight transactions land in awkward intermediate states that then need careful cleanup or idempo- tent reprocessing.

Distributed gateways are designed with the assumption that partial failure is normal, not exceptional. When a node be- comes unhealthy, traffic can be routed around it and processing continues on the remaining nodes. You typically see a short capacity dip and a latency bump while traffic shifts, caches refill, and the healthy nodes pick up the slack, but transactions still flow. In payments, where a few minutes of downtime turns into abandoned carts, angry merchants, and instant support escalations, that ability to degrade instead of stopping outright is often the strongest reason teams accept the extra engineering and operational overhead.

Scalability

Centralized gateways usually scale vertically, you keep the same core stack and buy headroom by adding CPU, memory, I/O capacity, and a bigger database tier, which can work well for quite a while when traffic is steady and the bottleneck is obvious. The trouble is that vertical growth is a blunt tool: upgrades get expensive, they are harder to execute safely, and sooner or later you run into ceilings where database contention, shared locks, and write amplification limit progress long before the application tier runs out of compute. Dis- tributed gateways go the other direction and scale horizontally by adding processing nodes and spreading requests across them, which matches how payment traffic actually behaves, daily peaks, flash sales, holiday surges, and bursts that no forecast sees coming. But "add nodes" only helps when the rest of the system scales too, because the gateway fleet still leans on shared dependencies like the transaction store, idempotency or dedupe state, tokenization and risk services, and the bank-facing connectors. If any of those become the choke point, extra nodes mostly increase contention and retry volume, making the platform louder without delivering real throughput.

Operational Complexity

Centralized gateways are usually easier to operate because behavior is concentrated in a small set of components. Mon- itoring stays straightforward, configuration changes have a tight blast radius, and debugging often reduces to one request path, one primary log stream, and one database to inspect. That simplicity matters when the team is small, when on-call practices are still maturing, or when payments are critical but not the company's main engineering focus.

Distributed gateways raise the operational bar. You have to reason about routing, uneven load across nodes, state synchronization, and what "correct" means when the system is degraded and retries are piling up. Observability needs to be stronger as well, typically structured logs with durable cor- relation IDs,

end-to-end tracing, and alerting that can separate node-local failures from shared-state consistency problems. That increases deployment and maintenance overhead and demands more judgment during incidents, but it pays back in resilience under partial failure and a scaling model that can absorb spikes without treating every traffic surge like an emergency.

Table II
Comparison Of Centralized And Distributed Payment Gateway Architectures

Metric	Centralized Arch	Distributed Arch
Transaction Latency	Low under moderate load	Higher due to coordination
Fault Tolerance	Low (single point of failure)	High
Scalability	Limited	High
System Complexity	Low	[illegible]
Operational Cost	Lower	Higher
Maintenance Effort	Simple	Complex

VIII. Limitations

These results come from a simulation meant to surface architectural trade-offs, so a production gateway will behave more messily than the model can capture. The setup inten- tionally leaves out organization-specific and network-driven factors such as bank-side processing delays, issuer throttling, proprietary risk scoring pipelines, and the variability intro- duced by third-party payment rails. In real deployments, those external dependencies often dominate tail latency and can change the failure story in ways a controlled environment will not capture. The synthetic workload generator approximates common request patterns, but real traffic has quirks, abrupt bursts, correlated retries, uneven merchant mixes, and region-specific behavior that can shift bottlenecks and change which component becomes the limiting factor. Several trends in latency and throughput are also tied to deliberate modeling choices, for example uniform network delay and node-level failure injection, and different assumptions (variable network jitter, upstream brownouts, per-issuer timeouts, or queue- backed async confirmation) could yield different curves and different breakpoints.

The distributed architecture here is treated as a concep- tual design choice, not a hardened production blueprint. Real distributed gateways typically need extra machinery to stay correct and recover cleanly, replication strategies, strict idempotency enforcement, retry deduplication, and explicit handling of partial commits across components [7], [8]. That machinery is not free. It adds overhead in the steady state, and it can also change how failures show up, especially once you move past clean "node down" events into the failures that actually cause trouble in production, slow or uneven nodes, inconsistent replicas, and backpressure rippling through queues and shared stores. For that reason, the comparison should be read as guidance on expected architectural behavior and trade-offs, not as a universal performance claim that would transfer unchanged across every implementation and operating environment.

Table III
FAILURE SCENARIOS CONSIDERED IN THE STUDY

Scenario	Expected Behavior
Central gateway failure	Service disruption until recovery
Central database failure	Transaction processing halted
Single node failure (distributed)	Requests rerouted to healthy nodes
Load balancer failure	Degraded routing, possible disruption

IX. Conclusion

Centralized and distributed payment gateway designs can support the same checkout workflow, but they behave differently once you care about failure modes and growth. tralized gateway remains appealing because it is oper- lly clean, easier to secure and audit, and often delivers latency as long as traffic stays within the comfortable limits of the gateway instance and its backing database. For many products, especially with predictable volume and a small team running the platform, that simplicity is not a compromise, it is exactly what makes the system workable.

Distributed gateways make the most sense when uptime is non-negotiable and growth is something you can see com- ing, not a slide-deck forecast. Spreading transaction handling across multiple nodes lowers the chance that a single bad host or overloaded component takes checkout down, and it gives a straightforward path to adding capacity as demand climbs. You pay for that with coordination work, more moving parts, and a higher bar for observability and operational discipline. The bigger takeaway is that the architecture choice is rarely about one headline number, it comes down to traffic shape (steady versus spiky), the real business cost of

downtime, how much failure the product can tolerate without losing revenue or trust, and what the team can realistically operate well for years.

X. Future Work

Future work should test hybrid gateway designs that keep a small amount of centralized control (policy, configuration, auditing, or a single source of truth for settlement) while pushing request intake and authorization handling into hori- zontally scaled, mostly stateless nodes, then measure whether that split preserves operational clarity without the "one bad day takes everything down" failure mode. The evaluation should also move beyond clean synthetic workloads by introducing production-like traffic spikes, network jitter, regional latency differences, and correlated retry storms that appear when issuers or processors slow down, since those conditions often decide real tail latency and correctness risk. A more realistic study should include the performance footprint of security controls such as tokenization, HSM-backed key operations, anomaly scoring, and fraud checks under high concurrency, and it should tie the technical results to economics by modeling infrastructure cost and energy usage alongside throughput and availability targets across different traffic profiles.

References

[1] K. C. Laudon And C. G. Traver, E-Commerce: Business, Technology, Society, 14th Ed. Pearson, 2018.
[2] D. Humphrey, "Payment Systems: Principles, Practice, And Improve- Ments," Journal Of Banking And Finance, Vol. 25, No. 8, Pp. 1493–1511, 2001.
[3] M. Greenstein And T. Feinman, Electronic Commerce: Security, Risk Management And Control. Mcgraw-Hill, 2016.
[4] S. G. Rao And S. R. Bhat, "A Study On Secure Online Payment Systems," International Journal Of Computer Applications, Vol. 96, No. 5, Pp. 21– 26, 2014.
[5] A. S. Tanenbaum And M. V. Steen, Distributed Systems: Principles And Paradigms, 2nd Ed. Pearson, 2017.
[6] M. P. Papazoglou And W.-J. Van Den Heuvel, "Service Oriented Architec- Tures: Approaches, Technologies And Research Issues," The Vldb Journal, Vol. 16, No. 3, Pp. 389–415, 2007.
[7] J. Gray And A. Reuter, Transaction Processing: Concepts And Tech- Niques. Morgan Kaufmann, 1993.
[8] M. T. Özsu And P. Valduriez, Principles Of Distributed Database Systems, 3rd Ed. Springer, 2011.
[9] R. Buyya, C. Vecchiola, And S. T. Selvi, Mastering Cloud Computing. Mcgraw-Hill, 2013.
[10] P. Mell And T. Grance, "The Nist Definition Of Cloud Computing," National Institute Of Standards And Technology, Tech. Rep., 2011.
[11] V. Paxson, "End-To-End Internet Packet Dynamics," Ieee/Acm Transac- Tions On Networking, Vol. 7, No. 3, Pp. 277–292, 1999.