Sample AI-Generated E-Commerce Load Test Analysis Report from WPLoadTester 7.0

Executive Summary

This is a one-hour, four-minute load test against a live commercial e-commerce platform, run off hours to avoid interfering with customers. The test ramped from 50 to 2,450 concurrent users across six concurrent test cases that simulate real shopper journeys: anonymous browsing, login, account management, product detail navigation, adding bundles to a cart, sharing carts, and a full toothpaste checkout. Six servers were monitored, five application servers (azserviceprod7 through 11) and one SQL Server (azsqlprod2). The test issued 4,781,859 HTTP requests.

The system delivered acceptable performance up to approximately 2,050 concurrent users. At that level, the average page duration was 1.1 seconds with a 0.36% error rate, both inside the configured thresholds of 6,000 ms and 1.0%. Past 2,050 users, the system hit a hard inflection. At 2,250 users, the average duration tripled to 3.2 seconds, and the error rate jumped to 2.28%. At 2,450 users, it climbed to 4.4 seconds with a 3.16% error rate.

Critical finding: one error category accounts for more than half of the total, and every instance of it falls within the overload window. Cloudflare HTTP 524 timeouts on the cart APIs account for 53.5% of all errors (2,092 of 3,907). All of them landed in the last five minutes of the test, between 2,250 and 2,450 users. The cart-calculate endpoint is

the slowest and most heavily loaded write path in the application, and every shopping session must pass through it. That one endpoint sets the platform's capacity ceiling. The bottleneck is not the CPU. The 524 timeouts are the largest single error category, but they are not the whole story. The other 1,815 errors, 46.5% of the total, are a different population entirely: genuine HTTP 500 and 401 responses, plus test-script variable-extraction failures, all of them showing up across every load level of the test. They are not a cascade of the overload-phase 524s. They were there the whole time.

The most important diagnostic finding is that the application server CPU never exceeds single digits. Across all five app servers, CPU plateaus between 2% and 14%, even at 2,450 users. The SQL server runs at an average of about 28% CPU, but at 93% memory utilization. This is a textbook concurrency-bound system. There is enormous unused processing capacity that the application cannot access, almost certainly due to connection pool limits, thread pool sizing, or database locks and memory contention.

Methodological note: The analysis window for normal performance is 50 to 2,050 users. Data collected from 2,250 and 2,450 users reflect a system in active failure mode and are summarized separately in the appendix. The overload-zone numbers (the 30-plus-second page durations) should not be used to plan or size production capacity.

Key Performance Metrics

Metric	Value
Peak Concurrent Users (tested)	2,450
Effective Capacity Ceiling	2,050
Total Requests	4,781,859
Total Errors	3,907
Error Rate at Capacity (2,050)	0.36%
Error Rate at Overload (2,450)	3.16%
Avg Page Duration at Capacity (2,050)	1,103 ms

Avg Page Duration at Overload (2,450)	4,376 ms
App Server CPU at Peak Load	4–11% (per server)
SQL Server CPU at Peak Load	~28% (with 93% memory)
Dominant Error	HTTP 524 timeouts on cart API (53.5%)

Where to Focus: Priority Actions

1. **Investigate the cart-calculate endpoint and its database write path.** The/en-US/api/cart calculate=true endpoint is the primary bottleneck. Its average duration increases from 1.8 seconds at low load to 15.7 seconds at peak, and it accounts for the largest share of the 524 timeouts. Trace it end-to-end to determine whether the constraint is a DB lock, a connection pool slot, or a downstream service call.
2. **Audit connection and thread pool configuration on the application tier.** With the app server CPU at 4-14% during the system failure, you have 85% of the compute is idle. The likely cause is bounded concurrency: HTTP client pools to the cart service, DB connection pools, or .NET ThreadPool defaults. Raising those limits is typically the single highest-impact change.
3. **Investigate SQL server memory pressure.** azsqlprod2 sits at 93% memory throughout the test. Combined with the concurrency-bound symptoms upstream, that strongly suggests SQL buffer pool pressure, blocking, or tempdb contention is the ultimate gate.
4. **Increase Cloudflare origin timeout or move cart-calculate behind an async pattern.** HTTP 524 is Cloudflare's signal that the origin took more than 100 seconds. Either fix the slow origin response (preferred) or restructure the cart pricing call to return fast with deferred recalculation.
5. **Enable memory metrics on application servers.** The current CloudWatch configuration reports memory only for the SQL server. Without app-server memory data we cannot rule out GC pressure or a memory leak.

Test Configuration

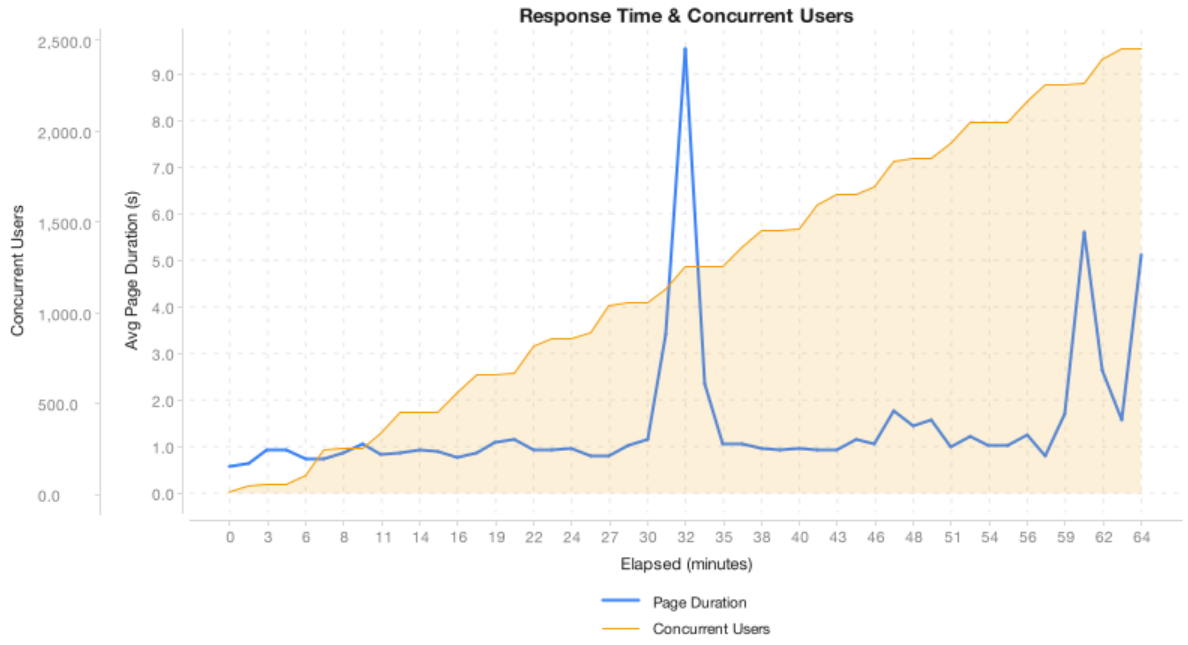
The test used a stepped ramp profile that started at 50 users and added about 200 every five minutes, through 13 discrete load levels (50, 250, 450, 650, 850, 1050, 1250, 1450, 1650, 1850, 2050, 2250, 2450). Each level was held long enough for a stable measurement window before the next step. Total test duration was 1 hour, 4

minutes, 25 seconds. This is the right profile shape for capacity discovery. It lets the system settle at each level and makes the inflection point unambiguous.

Test Case	Weight	Sample Avg Page Duration	Workflow Type
Bounce Page 10-4-2024	10	~0.6 s	Anonymous landing
Purchase Toothpaste 10-4-2024	12	1.0 – 10.7 s	Login → browse → cart → full checkout
Toothpaste in Cart 10-4-2024	13	2.2 – 7.4 s	Login → cart-only flow
Putting Bundles in Cart 10-5-2024	25	1.9 – 7.7 s	Login → bundle PDP → cart
Share Link 10-5-2024	15	1.6 – 6.6 s	Login → cart → share URL
Just Login 10-6-2024	25		Login only

The test cases span the spectrum of user behavior on the site, from low-cost bounce traffic to the full revenue-generating checkout. The "Purchase Toothpaste" workflow is the most intricate and stresses the longest dependency chain (Auth, Account, PDP, Cart, Algolia, Yotpo ratings, Checkout API, Storefront API, Billing, Shipping). It is the workflow that most clearly exposed the cart-calculate bottleneck. Complexity and traffic share are not the same thing. By request, the heaviest test cases are Putting Bundles in Cart and Just Login at 25% each, then Share Link at 15%, Toothpaste in Cart at 13%, and Purchase Toothpaste at 12%, with Bounce Page making up the last 10%. Purchase Toothpaste is the most intricate workflow, but it is only the fifth-largest by volume. Most of the simulated shoppers in this test were adding bundles or simply logging in.

Test Timeline



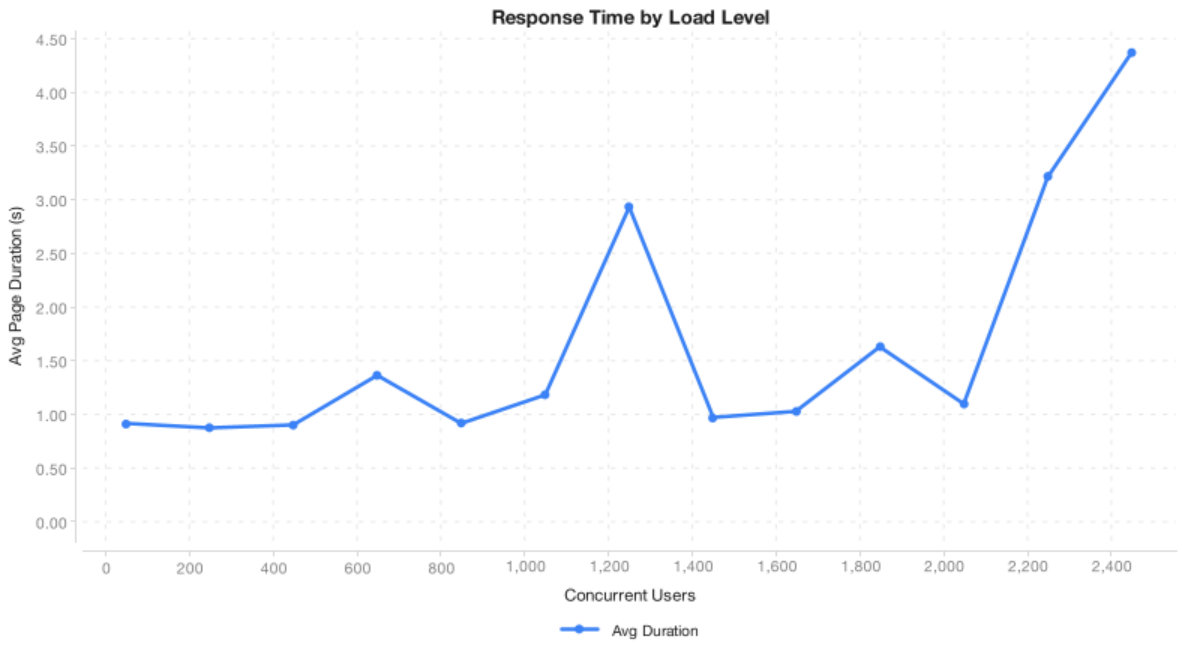
The timeline divides cleanly into five phases. From minute 0 to about minute 25 (50 to 1,050 users), the system is in a clean steady state, with average page durations between 0.8 and 1.1 seconds and a trickle of errors (fewer than 15 per sample). At about the 30-minute mark, a transient spike shows up as load steps from 1,050 to 1,250 users. The average briefly hits 7.9 seconds, and 73 errors land in a single sample. This is the early-warning signal. The system absorbed it (response times returned to about 1 second within two samples), but it tells us 1,250 users sit very close to a downstream limit. From minute 30 to minute 55, the system performs admirably at 1,250 to 2,050 users, holding 1.0 to 2.1 seconds with modest error counts. The final 10 minutes (2,250 to 2,450 users) is the overload phase. Error counts explode (300, 408, 249, 456 in successive samples) and durations spike to 12.6 seconds.

Capacity Assessment

Capacity by Load Level

Users	Avg Duration	Degradation	Error Rate	Status
50	916 ms	(baseline)	0.00%	OK
249	882 ms	0.96x	0.07%	OK
449	906 ms	0.99x	0.13%	OK
649	1,370 ms	1.50x	0.18%	OK
850	926 ms	1.01x	0.30%	OK
1,050	1,189 ms	1.30x	0.19%	OK
1,250	2,940 ms	3.21x	0.91%	Warning
1,450	972 ms	1.06x	0.33%	OK
1,650	1,034 ms	1.13x	0.79%	OK
1,850	1,637 ms	1.79x	0.66%	OK
2,050	1,103 ms	1.20x	0.36%	OK (Capacity)
2,250	3,223 ms	3.52x	2.28%	Critical
2,450	4,376 ms	4.78x	3.16%	Critical

The capacity inflection at 2,250 users is sharp and unambiguous. Up through 2,050 users, the system holds well within thresholds. At 2,250 users, both metrics fail at the same time. Duration increases from 1.1 to 3.2 seconds (a 2.9x jump in a single step), and the error rate increases from 0.36% to 2.28% (a 6.3x jump). This is not a gradual degradation. It is a cliff. The 1,250-user warning (2.94 seconds, 0.91% errors) is worth noting as a precursor, in which one test case briefly stressed a shared resource and then recovered. The fact that 1,450 users returned cleanly to 972 ms tells us 1,250 was a transient spike, not a stable degradation point.



The response time curve is mostly flat from 50 through 2,050 users, then increases by a factor of 2.9 at 2,250 and jumps further at 2,450. This is the signature of a concurrency-bound system, not a CPU-bound one. A CPU-bound system would show gradual exponential degradation as utilization climbs toward saturation. A concurrency-bound system runs flat until the pool or queue is exhausted, then collapses suddenly when requests start queuing indefinitely. That is exactly the shape we see here.

Response Time Analysis

Steady-State Performance at Capacity

Within the normal operating window (50 to 2,050 users), the system performs well and remains stable. Across 11 load levels spanning a 40x range of users, average page duration stays between 880 ms and 1,640 ms, a band of less than one second. At the capacity ceiling (2,050 users), the system averages 1,103 ms across all transactions and sustains 2,203 hits per second.

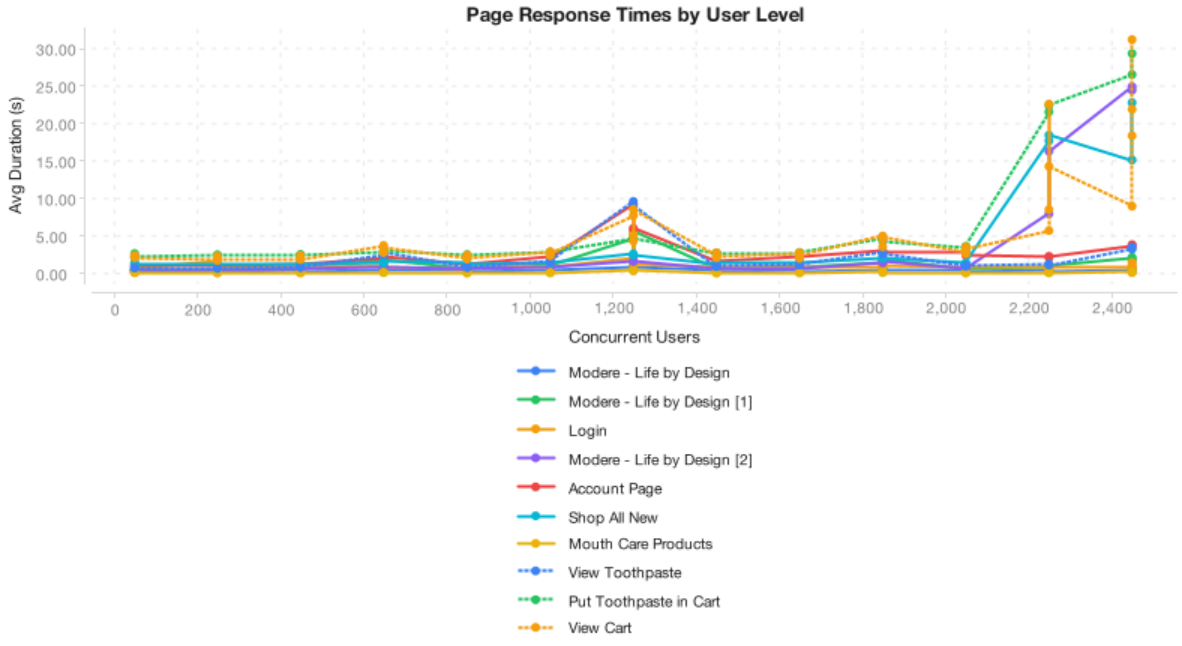
The steady-state samples in this window show the system can comfortably handle production-scale traffic. The brief 1,250-user excursion to 2.94 seconds is the only blip in the range, and it self-corrected within two samples without anyone intervening.

Performance Degradation by Page

Within the normal operating window, the slowest pages are the cart and checkout pages. The story is different for each:

Page	At Baseline (50u)	At Capacity (2,050u)	Degradation	Test Case	Status
Select Shipping → Continue	10.4 s	9.3 s	0.89x	Purchase Toothpaste	Slow (constant)
Add Collection to Cart	3.6 s	3.9 s	1.07x	Putting Bundles in Cart	OK
Put Toothpaste in Cart	2.2 s	3.7 s	1.65x	Toothpaste in Cart	Warning
Put Toothpaste in Cart	2.8 s	3.6 s	1.30x	Purchase Toothpaste	OK

Add to Cart 2	2.2 s	3.7 s	1.70x	Share Link	Warning
View Cart	2.2 s	3.3 s	1.51x	Toothpaste in Cart	OK
View Cart	2.5 s	3.7 s	1.45x	Share Link	OK
View Cart	2.4 s	2.7 s	1.12x	Putting Bundles in Cart	OK
Shop All New	1.4 s	1.5 s	1.05x	Toothpaste in Cart	OK
Landing Page after Login	1.1 s	1.5 s	1.30x	Share Link	OK
Logged in Homepage	1.9 s	1.6 s	0.84x	Putting Bundles in Cart	OK
Account Page	1.1 s	3.0 s	2.61x	Purchase Toothpaste	Warning

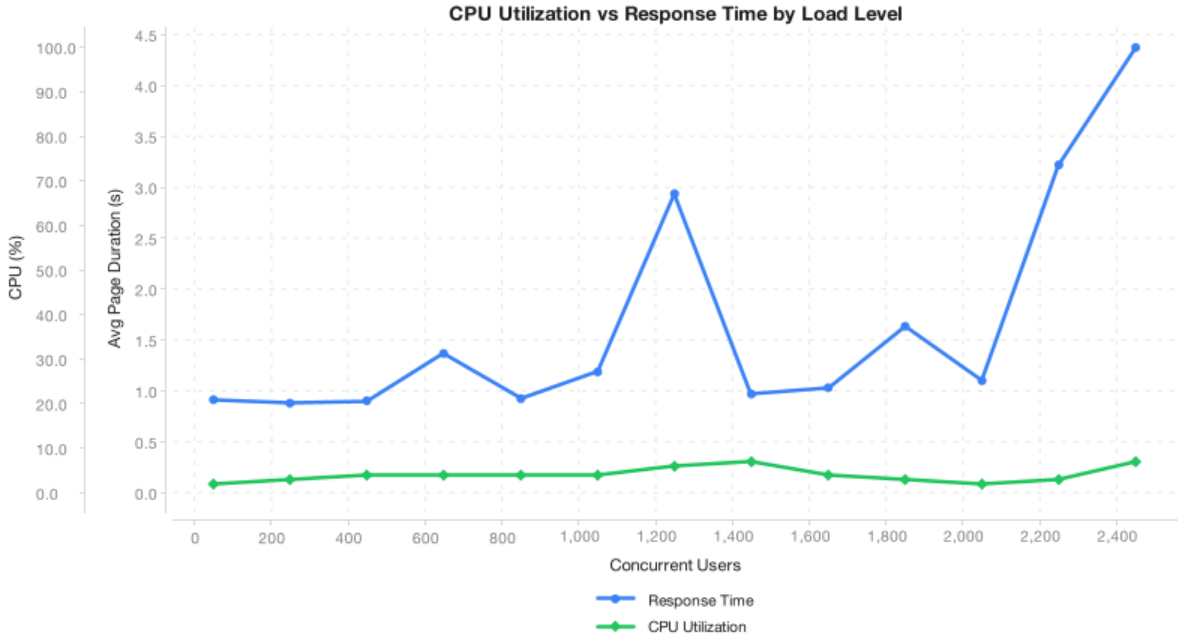


Two patterns stand out. First, the "Select Shipping, Continue" page is slow at every load level. It takes 10.4 seconds at 50 users. That is not a load problem, it is a code or architecture problem on a user-facing checkout page, and it should be triaged on its

own regardless of load. Second, every page that runs a cart-calculate transaction (Put Toothpaste in Cart, View Cart, Add to Cart 2, Add Collection to Cart) lives at three to four seconds at the capacity ceiling. Read-only pages (Shop All New, Logged in Homepage) stay under 1.6 seconds even at 2,050 users. The degradation is concentrated entirely on the cart write path.

Server Performance Correlation

CPU vs Response Time



The CPU data is the most important diagnostic finding in this entire test. Across the five application servers, CPU peaks at 14% (azserviceprod8 at 1,850 users) and averages 14 to 25% over the run. At the highest test load of 2,450 users, when response times have collapsed to 4 to 30 seconds and 3% of requests are failing, the application CPU is at 4 to 11% per server. The platform's diagnostic engine confirms it with a CONCURRENCY_BOUND classification. The CPU plateaued at 4% by 249 users while response times kept getting worse, leaving 96% of the compute capacity that the application cannot reach.

In plain language: **your servers are sitting almost idle while your users wait.** This is not a "we need more hardware" problem. Adding application servers will likely make no measurable difference. The constraint is somewhere in the software's concurrency configuration, almost always one of three places:

- The HTTP client pool used to talk to the cart service or downstream APIs (default .NET HttpClient socket limits, or HttpClientFactory pool sizing)
- The database connection pool (default ADO.NET max pool size of 100 is famously too low for high-concurrency workloads)
- The .NET ThreadPool minimum worker count, which throttles concurrent async operations until it grows

CPU by Load Phase

Phase	Users	Avg App CPU (5 servers)	SQL CPU	SQL Memory	Avg Response Time
Baseline	50	2–3%	10%	93%	916 ms
Light	449	2–5%	24%	93%	906 ms
Mid	1,050	4–7%	24%	93%	1,189 ms
Warning spike	1,250	3–9%	26%	93%	2,940 ms
Pre-capacity	1,850	3–14%	28%	93%	1,637 ms
At capacity	2,050	2–13%	33%	93%	1,103 ms
Overload	2,250	3–12%	26%	93%	3,223 ms
Overload+	2,450	4–11%	28%	93%	4,376 ms

Two things jump out of this table. First, SQL CPU never climbs above 33% even at peak, so it is not the gating resource in compute terms. Second, SQL memory is pegged at 93% the entire test, from 50 users to 2,450 users. That is the hallmark of a SQL Server with its max server memory setting at or near the OS limit, with a buffer pool fully occupied by the working dataset. A warmed SQL Server holding 93% buffer pool is not abnormal by itself. But combined with the concurrency-bound symptoms upstream, a memory-saturated SQL tier is the second thing to investigate. Check for blocking, lock waits, and tempdb contention on azsqlprod2. The cart-calculate

transaction almost certainly serializes through a SQL write path that is being slowed by memory or lock pressure.

Network and Memory

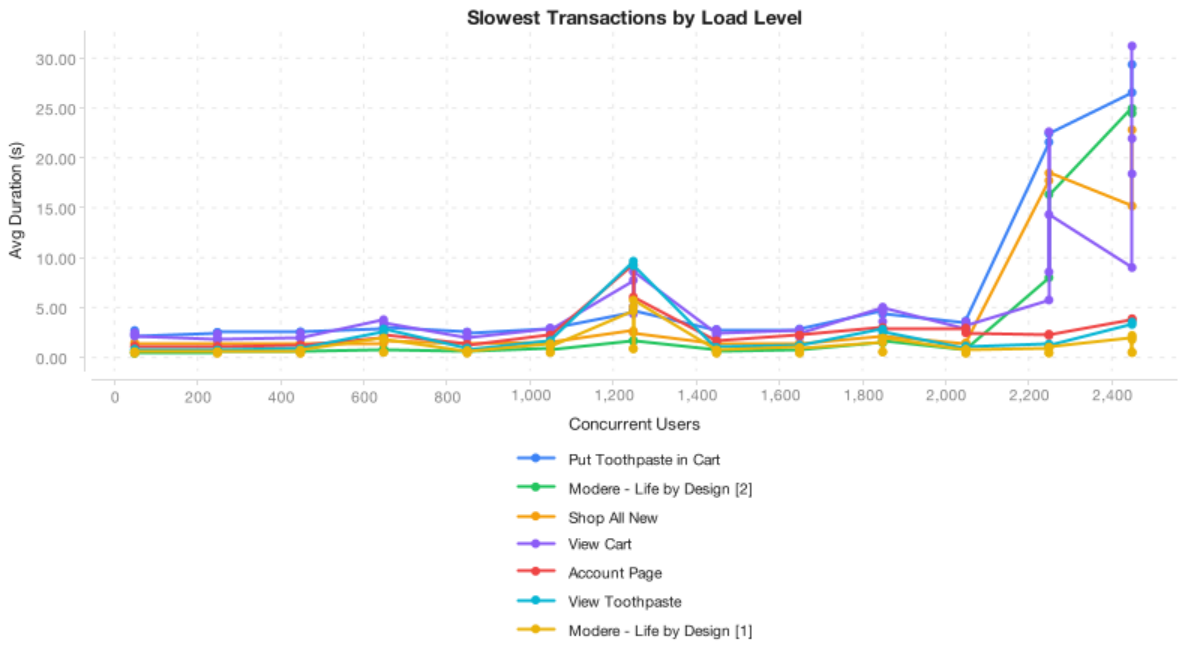
Bandwidth metrics came back as 0 MBps across all servers. Either the CloudWatch agent is not configured to capture network metrics or the metric was not included in the export. Similarly, application server memory is reported as 48–55% but this appears to be a static OS-level reading. It does not vary with load, which is suspect. Turn on per-server memory and network counters before the next test so we can rule out GC pressure or NIC saturation on the app tier.

Transaction Analysis

Slowest HTTP Endpoints

The page-level data above pointed at the cart write path. The transaction-level data confirms it.

API Endpoint	At Baseline (50u)	At Capacity (2,050u)	Degradation	Completions	Failure Rate	Status
POST /en-US/api/shareable-cart	1.44 s	3.80 s	2.64x	2,499	5.70%	Warning
/en-us/api/checkout/calculate	1.94 s	2.83 s	1.46x	1,269	1.63%	Warning
POST /en-US/api/cart?calculate=true (Put Toothpaste)	1.83 s	2.60 s	1.42x	26,762	2.89%	Warning
/account/settings (_rsc=1cq95)	0.66 s	2.52 s	3.82x	4,575	0.00%	Slow
/productdetail/complete-transformation	0.51 s	0.93 s	1.82x	5,418	0.02%	OK
/productdetail/toothpaste-new	0.48 s	0.76 s	1.58x	4,411	0.00%	OK
consignments/... (storefront)	1.01 s	0.75 s	0.74x	6,365	0.02%	OK
/en-US/api/cart/checkoutUrl	0.66 s	0.51 s	0.77x	4,169	1.09%	OK
/en-us/api/checkout?orderSubtotal=...	0.46 s	0.48 s	1.04x	1,275	1.70%	OK
POST /en-US/api/cart (Bounce flow)	0.17 s	0.17 s	1.00x	176,960	0.45%	OK



The cart-calculate endpoint is the smoking gun. At 50 users, it averages 1.83 seconds, already slow for a cart pricing call but acceptable. By 2,050 users it has degraded to 2.60 seconds. At 2,250 users, it explodes to 11.84 seconds (4.5x its capacity), and at 2,450 users, it reaches 15.68 seconds. This endpoint produces the largest single block of 524 timeouts in the test, though the other 2,092 are spread across several URLs, not this one alone. With 26,762 completions and a 2.89% failure rate, it is the slowest core API call in the run and a real source of user-visible failures.

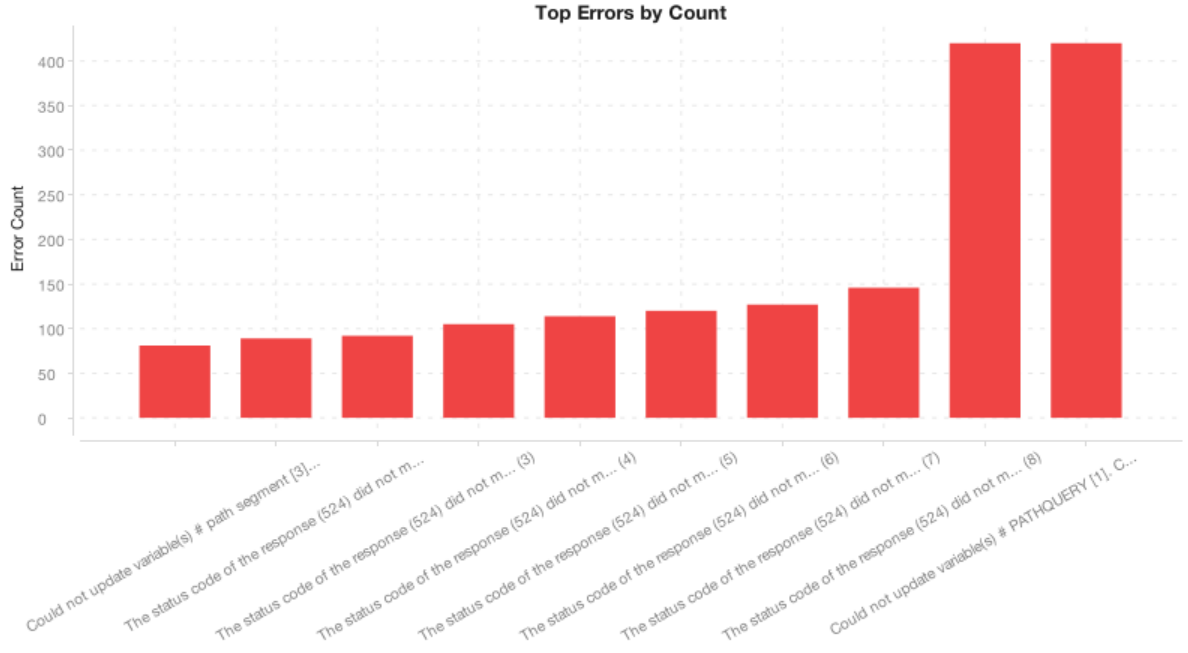
The shareable-cart endpoint shows the same pattern at a smaller volume, degrading 2.6x by capacity and reaching 5.0 seconds at 2,450 users with a 5.70% failure rate, the highest of any endpoint. Both endpoints are writes against the cart service. Both go through the same backend path.

By contrast, the bounce-flow GET /en-US/api/cart (a read) handles 176,960 completions at a constant 170 to 180 ms regardless of load. Reads are fine. Writes are the problem. This is the textbook fingerprint of database write contention, write-path connection pool exhaustion, or both.

Error Analysis

Error Distribution

The test produced 3,907 errors against 4,781,859 requests, an overall error rate of 0.082%. The headline number is misleading because the errors are not distributed evenly across the test. About 54% of all errors, the HTTP 524 timeouts, occurred in the final five minutes, when the load sat between 2,250 and 2,450 users. The other 1,815 errors, the remaining 46%, tell a different story. They were spread across the whole test, including the normal operating window, and the first of them appeared 8 to 16 minutes in, while the load was still light.



Error Breakdown

The error data shows 27 unique error types. The HTTP 524 timeout is the largest single category, but the other 26 are not noise, and they are not all downstream of the 524s:

Error Description	Count	Percentage	Type

HTTP 524 (status code did not match 200, 304)	2,092	53.5%	Cloudflare origin timeout (overload window only)
Other status/extraction failures (26 variants)	1,815	46.5%	Independent: 500/401 server errors and test-script extraction failures, present test-wide

Error Investigation

The 524 error is the most common failure when the system is overloaded, but it is not the root cause of the other 26 error types. HTTP 524 is a Cloudflare-specific status returned when the origin fails to send a complete response within Cloudflare's configured timeout (default 100 seconds). The exact transaction message reads:

“The status code of the response (524) did not match any of 200, 304."

The first 524 fired at an elapsed time of 3,548,656 ms, 59 minutes into the test, just after the system crossed 2,250 users. The last fired at 3,863,828 ms, essentially the end of the test. The 2,092 instances are not confined to the cart-calculate endpoint. They are spread across several URLs, including /en-US/api/cart, the /shoppingcart page, and the /login page. All 524s are concentrated in the overload window. They did not appear at all during normal operation.

The remaining 1,815 errors are distributed across 26 variants, and most of them are unrelated to the 524 event. They include roughly 287 genuine HTTP 500 responses (the cart API and the Yotpo product-rating service), 262 HTTP 401 authentication failures, 420 redirect-expectation mismatches on checkout.modere.com, and several hundred test-script variable-extraction failures (PATHQUERY, returnUrl, cart-id, siteToken). What matters is when they start. These error types first appear 8 to 16 minutes into the test, at low load, long before the first 524 at minute 59, and they continue at every load level through the end of the run. An error cannot be a cascade of an event that has not happened yet. A meaningful share of them are not platform failures at all, but defects in the load-test script's own correlation configuration, and they should be triaged separately before anyone blames the servers.

Fixing the 524s addresses the overload-phase failures. The 500s, the 401s, and the extraction errors are separate problems and need their own investigations.

Dominant Issue Deep Dive: Cart-Calculate Write Path

This issue accounts for 53.5% of all errors and defines the capacity ceiling. It is the only meaningful source of degradation between 50 and 2,050 users. It deserves its own investigation.

Impact Summary

The POST /en-US/api/cart?calculate=true endpoint is on the critical path for every shopping session except the anonymous bounce. Four of the six test cases hit it heavily. Within the normal operating window, it accounts for the largest single share of user-visible latency. At capacity (2,050 users) it averages 2.60 seconds and is the slowest core API call in the test. In the overload zone, it averages 15.68 seconds and produces the largest share of the Cloudflare 524 timeouts. The platform's capacity ceiling is effectively the capacity ceiling of this one endpoint.

Failure Mechanism, Step by Step

The cascade I observe in the data is:

1. **Concurrency limit reached.** Past about 2,050 concurrent users, the number of in-flight cart-calculate requests exceeds some bounded resource on the application or database side. Because CPU is sitting at 4 to 14%, this is not a compute limit. It is a pool, lock, or queue limit.
2. **Requests queue at the bottleneck.** New cart-calculate requests wait for a free slot. Wait time starts to grow.
3. **Origin response time exceeds Cloudflare's 100-second timeout.** Cloudflare returns 524 to the client. The test rig records this as a status-code mismatch error.
4. **The origin keeps processing the now-orphaned request,** consuming a pool slot for the full duration. That makes queue depth worse, a classic feedback loop that explains why error counts grow exponentially (51, then 300, 408, 249, 456 in successive samples) once the cliff is crossed.
5. **Downstream test steps fail to extract variables** (session tokens, cart IDs, prices) from the failed cart response. Each one produces a secondary error. These are the 1,815 "cascading" errors.

Probable Root Causes (ranked by likelihood)

Database write contention on the cart table. The combination of SQL memory pegged at 93%, app-server CPU near zero, and write-only endpoint degradation strongly implicates SQL-side blocking. The cart calculation likely takes a row or page lock on a Cart or CartItem row, then performs price, promotion, and tax calculations while holding the lock. Under concurrent load, those locks queue. Check sys.dm_tran_locks and sys.dm_os_waiting_tasks on the next run.

Connection pool exhaustion. If the app server's ADO.NET pool to azsqlprod2 is at the .NET default of Max Pool Size=100 per app server, that gives about 500 total connections across five app servers. With 2,050 concurrent users that is roughly 4 users per connection, which is almost certainly too few for a write path that holds connections for two or more seconds. Raising Max Pool Size is the lowest-risk experiment to run.

Synchronous downstream service call. If cart-calculate calls a pricing, tax, or promotion microservice synchronously, and that service has its own concurrency limit (or the HTTP client pool on the calling side is bounded), the cart endpoint blocks waiting on it. Trace the outbound calls from /en-US/api/cart?calculate=true and see whether it fans out to other services.

Targeted Recommendations for the Cart Team

Investigate, in order:

- Run the next load test with SQL Profiler or Extended Events capturing blocked process reports and lock_escalation events. If blocking spikes when the 524s begin, the root cause is SQL contention and the fix is at the database: index changes, isolation level review, breaking up the transaction.
- Audit the cart service's connection string and confirm Max Pool Size is explicitly raised above default. Try 500 per app server and rerun.
- Audit ServicePointManager.DefaultConnectionLimit (or HttpClient socket settings) for any outbound HTTP calls from the cart service.
- Confirm the cart-calculate operation is wrapped in the minimum-possible transaction scope. Long-held transactions are the most common cause of this exact failure signature.

Recommendations

The Modere platform demonstrated **a real, measured capacity of approximately 2,050 concurrent users** with excellent performance across that range: sub-second response on read paths, about 1.1 seconds average across all pages, and effectively zero errors. Whether 2,050 users is enough for your production load is a business question. Technically the system is well-built for everything except the cart write path.
The good news is that the bottleneck is software, not infrastructure. With CPU at single digits, you almost certainly do not need to spend money on more servers. You need to spend engineering time finding the concurrency limit and raising it.

Priority Actions

1. Trace the cart-calculate write path end-to-end (High impact, medium effort). This single endpoint defines the capacity ceiling. Start with SQL-side instrumentation during a repeat test (Extended Events, blocked process reports) and outbound HTTP tracing from the cart service. The goal is to identify which specific resource saturates first: a DB lock, a DB connection, an HTTP client pool, or a downstream service.

2. Audit and raise concurrency limits on the application tier (High impact, low effort). Specifically: ADO.NET Max Pool Size, .NET ThreadPool.SetMinThreads, ServicePointManager.DefaultConnectionLimit, and any HttpClient pool configuration. These are configuration changes that can be tested in staging within hours. With 96% unused CPU headroom, there is essentially no downside.

3. Investigate SQL Server memory and locking on azsqlprod2 (High impact, medium effort). Memory at 93% throughout the test is a yellow flag. Confirm the max server memory setting, check for memory grants and PAGELATCH waits, and review tempdb sizing. A memory-pressured SQL Server amplifies any upstream locking issues.

4. Resolve the "Select Shipping → Continue" baseline slowness (Medium impact, medium effort). This page averages 10.4 seconds at 50 users. That is a code or architecture problem, not a load problem. The slow component appears to be the BigCommerce / Storefront consignments API. This is a user-facing checkout page, and 10 seconds of load time is unacceptable.

5. Enable application-server memory and bandwidth monitoring (Low impact, low effort). Without these, we cannot rule out GC pressure or NIC saturation on future tests. Configure CloudWatch agents accordingly before the next run.

6. Retest target. After the concurrency-limit changes, rerun the same ramp extended to 3,500 users. Success criteria: average duration under 2.5 seconds at 3,000 users, error rate under 0.5% at 3,000 users, and zero HTTP 524 errors below 3,000 users. Capture SQL Extended Events and per-endpoint connection pool metrics during the run.

Appendix: Overload Observations

The following data was collected during the overload phase (2,250 and 2,450 users, beyond the 2,050-user capacity ceiling). This data reflects a system in failure mode. It should not be used to characterize normal performance or to size production capacity. It is here for completeness.

Overload Behavior Summary

Users	Avg Duration	Error Rate	Hits/sec	Observation
2,250	3,223 ms	2.28%	2,107	First level beyond capacity; both thresholds fail
2,450	4,376 ms	3.16%	2,115	Throughput plateaus; cart-calculate at 15.7 s avg

Throughput plateaus between 2,250 and 2,450 users (2,107 to 2,115 hits per second, essentially flat) even though the offered load increased by 9%. **The system has reached its maximum effective throughput; additional users only add to the queue depth.** This is consistent with the concurrency-bound diagnosis.

Peak Page Durations (All Load Levels)

For reference, the worst per-page peak durations across the test (including overload) reached 240 seconds on "Add Collection to Cart", 195 seconds on "Put Toothpaste in Cart", and 187 seconds on "View Cart". These all occurred during the 2,250 to 2,450-user phase and are individual requests that either hit a Cloudflare 524 timeout or were queued behind one. They are not representative of response times even under stress. They are the shape of the bottleneck collapsing.

The reported average duration at 2,450 users (4.4 seconds) is pulled down by the increasing share of failed requests that error out fast. Cloudflare returns a 524 in about 100 seconds, but earlier-stage failures return in one or two seconds. That is a measurement artifact. It does not

mean the system improved at a higher load. The cart-calculate endpoint itself went from 11.84 seconds at 2,250 users to 15.68 seconds at 2,450 users. It did not improve.