IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

Empirical Comparison of REST, WebSocket (WS) and SSE for Web Applications

Sahil Pathan
Department Of Computer Application
JSPM University, Pune, India

Abstract: *Web developers today face a genuine challenge in server communication. REST, WebSocket, and Server-Sent Events each solve the same fundamental problem — getting data from point A to point B — but they do so in different ways that produce measurably different outcomes under load. This study evaluated the performance of REST, WebSocket, and SSE under controlled concurrency conditions to determine which protocol is most suited for real-time web applications at scale. To address this, we built a containerised measurement tool containing three Node.js server microservices — one per protocol — and a dedicated load- generation mechanism, orchestrated through Docker Compose so that anyone can reproduce the experiment with a single command. The load-generation mechanism spawns between 10 and 1,000 concurrent virtual clients, runs each scenario for 10 seconds, and records average message latency, aggregate throughput, CPU load, and heap memory consumption. The numbers tell a reasonably clear picture: WebSocket maintained the lowest and most stable latency throughout, while SSE delivered competitive throughput with surprisingly modest CPU overhead. REST held its own at low concurrency but started showing its per-request cost as client counts climbed. These findings are not particularly surprising in hindsight, but having concrete, reproducible figures — rather than intuition — is the point.*

Keywords: REST, WebSocket, Server-Sent Events, real-time web, benchmarking, latency, throughput, Docker, Node.js

I. INTRODUCTION

When you work on a stock-trading dashboard, a live sports scoring app, or even a moderately active chat interface, you are observing the outcome of a protocol decision made somewhere in that system's design history. The choice of how the server and client exchange data is not merely a low-level detail — it shapes response time, server load, and ultimately whether the application feels alive or sluggish.

For most of the web's history, HTTP's request-response model was the only realistic option. Browsers requested pages from the server and server replied, the interaction ended. This worked fine when 'dynamic content' meant content that changed between page loads. When applications began tracking prices in real time or pushing notifications the moment they were generated, the mismatch between the stateless HTTP model and these continuous data needs became difficult to paper over. Developers began to use polling loops and long-polling hacks, both of which carry meaningful costs in unnecessary HTTP overhead and complexity in connection management.

Three approaches have since emerged as the main options for web applications with real-time or near-real-time requirements. REST — still dominant in API design — relies on HTTP request-response, which means the client must ask for new data each time it wants it. WebSocket upgrades an HTTP connection into a persistent, full-duplex channel over TCP, letting either end send messages at will. Server-Sent Events takes a lighter approach: HTTP, but with the response body left open so the server can push updates without the client ever re-asking.

Each of these three has a clearly articulated set of claimed advantages, but published comparisons tend to be qualitative or narrow in scope. For this study, we wanted numbers: latency distributions, throughput under concurrent load, and the CPU and memory usage each protocol imposes on the server process. Rather than relying on existing tooling, we built a

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 8, May 2026 Impact Factor: 8.2

small benchmarking platform from scratch — four Docker containers, three of which implement the protocol servers, one of which drives the load and captures measurements. The platform is deliberately simple, which is a strength: there are few moving parts to introduce confounding variance.

We ran the tool against five concurrency levels spanning 10 to 1,000 simultaneous virtual clients and collected results for all three protocols at each level. The rest of this paper describes what we built, how we ran the experiments, and what we found. Section 2 situates the work within existing literature. Section 3 provides a concise technical background on each protocol. Section 4 documents the testing tool architecture and implementation in enough detail that the reader could reproduce or extend it. Section 5 covers experimental methodology. Section 6 presents results and analysis. Sections 7 and 8 address limitations and draw conclusions.

II. RELATED WORK

The problem of efficiently pushing data from servers to browsers is old enough to have its own name — Comet — which dates to a 2006 blog post by Alex Russell. The earliest systematic academic treatment of the associated techniques appeared in the RFC-track work by Loreto et al. [1], which catalogued the failure modes and operational pitfalls of long-polling and chunked streaming. Their framing was more cautionary than empirical: they were mainly trying to get implementers to avoid well-known mistakes rather than quantify performance. Still, it remains a useful reference point for understanding why WebSocket and SSE were developed as deliberate alternatives rather than incremental improvements.

Pimentel and Nickerson [2] took a more developer-facing angle, looking at the HTML5-era APIs — EventSource and the WebSocket JavaScript API — and evaluating them for practical suitability in real-time web applications. Their work appeared around the same time as RFC 6455 [3], which formally specified the WebSocket wire protocol and settled a period of competing browser implementations. Neither study provided controlled latency or throughput benchmarks in the modern sense; they were assessing feasibility rather than performance.

The microservices era brought renewed interest in protocol performance, though the focus shifted somewhat. Taibi and Lenarduzzi [4], in their study of microservice anti-patterns, noted in passing that synchronous REST calls between services created measurable bottlenecks in high-throughput pipelines, but their paper was not about protocol comparison per se. More directly relevant is Wang et al. [5], who compared REST with gRPC under concurrent load. Their finding — that binary framing in gRPC produced substantially lower latency than HTTP/1.1 text serialisation at high message rates — is a useful data point for our study, because it raises the question of whether the text-based overhead of REST disadvantages it compared with WebSocket in similar conditions.

Probably the closest prior work to what we did here is Souza et al. [6], whose micro-benchmark compared HTTP polling, long-polling, SSE, and WebSocket in a browser context. They found that push-based protocols reduced server-side bandwidth consumption dramatically at high message rates, which is consistent with our throughput findings. However, they did not separately measure CPU and memory, and their test mechanism was not containerised, making their results harder to replicate.

The Node.js platform itself has been studied from a concurrency perspective by Chaniotis et al. [7], whose benchmark demonstrated that the single-threaded event loop model handles large numbers of concurrent I/O-bound connections well, performing comparably to multi-threaded servers on workloads dominated by waiting rather than computation. This matters for interpreting our results: the performance advantage of persistent connections in WebSocket and SSE is partly a function of Node.js's ability to multiplex many idle connections cheaply. A thread-per-connection server might show a different profile. Noor et al. [8] extended similar analysis to cloud and mobile contexts, noting that connection overhead becomes proportionally more significant as network latency increases — a point that suggests our Docker-internal results likely underestimate the real-world latency penalty REST pays at scale.

Fielding's dissertation [9] remains the canonical description of REST's architectural constraints, and it is worth noting that Fielding was explicit that REST is not necessarily optimal for low-latency, high-frequency communication — it was designed for the web as a document distribution system, not as a messaging platform. The WHATWG

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 8, May 2026 Impact Factor: 8.2

specification for Server-Sent Events [10] is the definitive technical reference for SSE semantics, including the retry and event-ID mechanisms that make it robust under intermittent connectivity. Both documents inform the protocol characterisations in Section 3.

III. PROTOCOL BACKGROUND

A. REST over HTTP

REST, in Fielding's original formulation [9], is an architectural style — a set of constraints on how components of a distributed hypermedia system interact. The constraint most relevant to performance is statelessness: each request from client to server must contain all the information needed to process it, and the server holds no session state between requests. This makes REST services easy to cache and load-balance, but it also means that every data retrieval requires a complete HTTP round trip.

In practice, a REST API call over HTTP/1.1 involves: a TCP connection (or reuse of a keep-alive connection), HTTP request serialisation including headers that can easily exceed 200 bytes even for simple requests, server-side request parsing and routing, application logic, response serialisation, and network transit in both directions. For a client that needs data every 100 milliseconds, the HTTP header overhead alone can dwarf the payload, and the event loop on the server must process each incoming request individually rather than simply writing to an already-open stream.

None of this makes REST a poor choice in general. Its stateless design integrates naturally with CDNs, API gateways, reverse proxies, and every monitoring and debugging tool developed over the past 25 years. Rate limiting, authentication, and response caching are all first-class HTTP features. For workloads where data is fetched infrequently or where the request-response model is semantically appropriate — CRUD operations on resources, for example — REST is the obvious choice and its overhead is irrelevant. The problems surface specifically when applications try to simulate real-time delivery by polling at high frequency, at which point the per-request overhead accumulates into something meaningful.

B. WebSocket

The WebSocket protocol [3] was designed explicitly to address the limitations of HTTP for bidirectional, low-latency communication. It begins with an HTTP/1.1 Upgrade handshake: the client sends an Upgrade: websocket header with a randomly generated security key; the server replies with HTTP 101 Switching Protocols and a computed accept token derived from that key. From this point, the TCP connection is no longer an HTTP connection — both sides may send framed messages at any time, in either direction, without any further request-response cycle.

The frame format is compact. A WebSocket frame header is between 2 and 14 bytes, depending on payload length and whether masking is applied. Client-to-server messages must be masked per the RFC, adding a small computational cost; server-to-client messages are unmasked. For the 40–80-byte JSON payloads typical in notification-style applications, this framing overhead is below 20% of the message size — substantially lower than the per-request HTTP overhead of REST polling.

The persistent connection also eliminates a specific TCP performance issue: slow-start. Every new TCP connection begins with a conservative initial congestion window and expands it as packets are acknowledged successfully. For REST polling, this slow-start cycle repeats with every new connection unless keep-alive is configured correctly. WebSocket sidesteps this entirely; by the time the application starts sending data, the connection has already warmed up during the HTTP upgrade phase.

Infrastructure compatibility is WebSocket's main operational friction point. Older reverse proxies buffer HTTP responses rather than passing them through incrementally, which breaks WebSocket framing. Most modern infrastructure supports it correctly, but it is still a common source of deployment surprises. Horizontal scaling is also more complex than for REST: because connections are stateful, requests from the same client must reach the same server instance unless a shared message broker sits in between.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

C. Server-Sent Events

Server-Sent Events occupy an interesting position: they use plain HTTP, which means no special proxy configuration, but they keep the response body open indefinitely, which means the server can push data without the client re-requesting. The mechanism is standardised as part of the HTML Living Standard [10] and exposes an EventSource interface in browsers and compatible Node.js libraries.

The wire format is simple text: event fields separated by colons, events separated by blank lines. A typical event looks like data: {"message":"tick","timestamp":1234567890}\n\n. This is more verbose than a WebSocket frame, but because SSE runs over HTTP/2 in environments that support it, multiple SSE streams can be multiplexed over a single TCP connection, reducing the connection count at the cost of stream coordination overhead.

One feature that often goes underappreciated is SSE's built-in reconnection. The EventSource API will automatically re-establish a dropped connection and, if the server is sending event IDs, will resume from the last received event. WebSocket applications that need similar resilience must implement reconnection logic themselves. For applications where the data stream is a continuous state update rather than a sequence of discrete events — live sports scores, for example, where missing one update is acceptable because the next will overwrite it — SSE's reconnection behaviour matters less. For event logs or notification streams where no events should be lost, it becomes genuinely valuable.

The constraint that SSE is unidirectional — server to client only — is its most limiting characteristic. Applications that need the client to send data back to the server must do so via a separate HTTP request, which may or may not be a meaningful architectural inconvenience depending on the use case. For dashboards, alert systems, live feeds, and telemetry displays, the server-push-only model maps naturally onto the application's data flow.

Characteristics	REST(HTTP)	WebSocket	SSE
Data direction	Client → Server	Bidirectional(full-duplex)	Server→ Client only
Connection lifetime	Short-lived per request	Persistent (long-lived TCP)	Persistent (open HTTP response)
Underlying transport	HTTP/1.1 or HTTP/2	TCP (after HTTP upgrade)	HTTP/1.1 or HTTP/2
Per-message header cost	High (full HTTP headers)	Very low (2–14-byte frame)	Low (Chunked HTTP stream)
Proxy/CDN compatibility	Universal	Requires explicit support	Universal (Plain HTTP)
Browser auto-reconnect	N/A	Manual implementation required	Built-in(EventSource)
Bidirectional use	Via paired request	Native	Server push only
Typical use cases	CRUD APIs, Resource Fetching	Chat , Games , Interactive tools	Dashboards , feeds notification

Table 1: Side by side comparison of REST, WebSocket and SSE characteristics

IV. TESTING TOOL DESIGN AND IMPLEMENTATION

When we started thinking about how to run this comparison, the first question was whether to adapt an existing benchmarking framework — tools like k6, JMeter, or wrk — or build something purpose-specific. We settled on a custom tool for a straightforward reason: generic HTTP benchmarking tools do not natively handle WebSocket or SSE with the same measurement semantics, so adapting one would have required nearly as much code as writing our own. A custom tool also meant we could instrument exactly what we cared about — per-message latency computed from server-embedded timestamps — rather than relying on proxy measurements at the HTTP layer.

The resulting platform consists of four Docker containers managed by a single docker-compose.yml. Three containers implement protocol servers; the fourth runs the load-generation and measurement harness. The source is available at https://github.com/sahil7249/testing-tool-Docker-deployed-.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

A. Overall Architecture

Each server container is a Node.js process that generates a continuous stream of timestamped messages. The timestamp is embedded server-side at the moment the message is sent, not when it is scheduled. This is important: a timestamp set in setInterval() reflects the scheduling time, not the send time, and the two can diverge under event loop pressure. By generating timestamps inside the send path, we ensure that the latency computed at the client — Date.now() minus the received timestamp — measures the actual delivery delay rather than a combination of delivery delay and scheduling jitter.

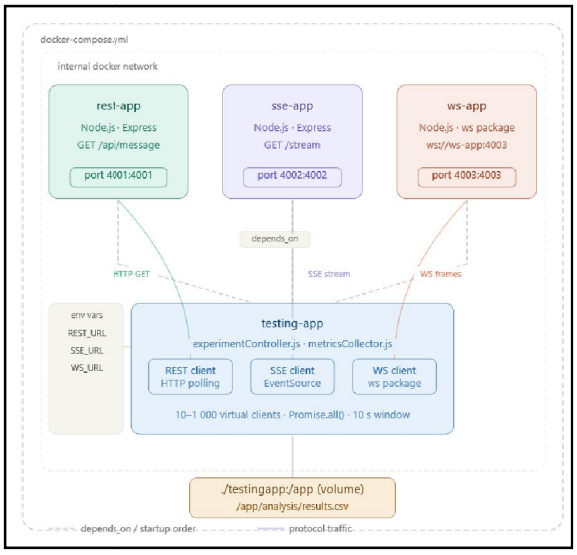

Fig 1. System Architecture – Four Container Docker Compose Topology

The testing container runs all three protocol experiments sequentially, not concurrently. Sequential execution was a deliberate choice to prevent the resource consumption of one protocol's clients from confounding the measurements of another. Within each protocol-and-user-count combination, all virtual clients run concurrently via Promise.all() for the full 10-second measurement window.

B. Server Implementations

B1. REST Server

The REST server is an Express.js application on port 4001. It maintains a counter variable that increments once per second via setInterval(). GET requests to /api/message receive a JSON body containing the current counter value and a timestamp generated at the moment the route handler executes. We chose to timestamp in the handler rather than in the interval because the route handler is what the network stack is actually responding to; any delay between the interval firing and the response being written would otherwise be invisible to our latency measurement.

B2. WebSocket Server

The WebSocket server uses the ws package, listening on port 4003. For each incoming connection, it registers a per-client setInterval() that fires every second and pushes a JSON frame. The interval is torn down on the close event, which is critical — without this cleanup, disconnected clients would leave zombie intervals accumulating in the Node.js event loop, inflating CPU measurements in later runs.

B3. SSE Server

The SSE server is another Express.js application, this time on port 4002. The /stream route responds with the three headers required by the EventSource specification — Content-Type: text/event-stream, Cache-Control: no-cache, and

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 8, May 2026 Impact Factor: 8.2

Connection: keep-alive — and then begins writing SSE-formatted events at one-second intervals. The data: prefix and double newline are mandatory parts of the wire format; omitting either causes the EventSource parser to silently discard events, which would produce misleading zero-latency measurements on the client side. We verified correct parsing early in development by running the client against a mock server with deliberately malformed events.

C. Client Load Generators

Each client module accepts a duration in milliseconds, runs for that duration, and resolves with two aggregated values: the total number of messages received and the cumulative latency sum. The controller divides these to produce per-client average latency and uses the message count to compute throughput.

The REST client is the most mechanically different from the other two. Because HTTP is request-driven, the client operates in a while loop, issuing one GET request, waiting for the response, computing latency from the timestamp in the response body, and then immediately issuing the next request. This means each virtual REST user is limited to one outstanding request at a time, which is a realistic representation of a browser-side polling loop but means the REST throughput measurements reflect sequential polling rather than pipelined requests.

The WebSocket client opens one connection, attaches an onmessage listener, and after the duration expires calls ws.close() before resolving. No explicit message-sending from client to server is needed — the server pushes on its own schedule. The SSE client follows the same pattern using the eventsource npm package, which provides an EventSource-compatible interface in Node.js. One subtlety: if the SSE connection has not received any messages by the time the timeout fires — which can happen if the server's one-second interval happens to align just past the timeout — the client resolves with latency: 0 and messages: 0 to avoid a division-by-zero, which the controller correctly handles by excluding zero-message runs from latency averages.

D. Metrics Collection

CPU and memory samples are captured using the systeminformation library, which reads from OS-level counters rather than Node.js process stats. We call getSystemStats() once per client spawn, inside the client-creation for loop, which means the CPU and memory readings accumulate as the client population grows. This is not a pre-test snapshot or a post-test average — it is a rolling average over the ramp-up phase. In hindsight, sampling at client spawn time means the early samples in a 1,000-user run reflect a much lighter load than the later ones, which somewhat compresses the reported average toward the low end. Future iterations of the tool should sample continuously in a separate timer and report the steady-state average once all clients are active.

E. Docker Compose Configuration

The depends_on relationships in the Compose file ensure that the testing container starts only after all three server containers are up. In practice this means the servers have already bound their ports and started their counter intervals before any client traffic arrives. Environment variables carry the service URLs into the testing container, so the client code has no hardcoded hostnames and the whole stack can be moved to a remote machine by simply adjusting the compose environment block.

V. EXPERIMENTAL METHODOLOGY

A. Variables

The experiment has one primary independent variable — the communication protocol, taking values REST, WebSocket (WS), and SSE — and one secondary independent variable: concurrent client count, taking values from the set {10, 50, 100, 500, 1000}. Together these define fifteen experimental conditions, each of which produces one data row in the output CSV.

The dependent variables are: message latency (milliseconds, client-side, averaged across all messages received by all clients in a run); throughput (total messages received across all clients divided by the 10,000 ms run duration, in

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

messages per second); CPU utilisation (percent of available logical cores, averaged across per-client-spawn samples); and memory usage (megabytes of used system memory, same averaging method).

B. Procedure

For each concurrency level, the controller runs REST first, then WebSocket, then SSE. Strict sequencing avoids cross-protocol interference: while REST clients are polling, no WebSocket or SSE connections are open, so the server for each protocol is handling only its own client population. Within a given run, the for-loop spawns clients one at a time (sampling system stats on each iteration) and pushes their promise into the clients array. Only after all client promises are created does Promise.all() resolve them concurrently, meaning the 10-second measurement window runs with all virtual users active simultaneously.

There is no artificial warm-up phase. Docker Compose's depends_on mechanism provides a basic readiness guarantee — the server processes are running and listening by the time the testing container starts — but V8 JIT compilation and OS page-cache warm-up are not accounted for. This is a limitation we acknowledge in Section 7.

C. Metric Definitions

Metric	Definition	Unit	Where measured
Avg. Latency	Mean of (client receive time – embedded server timestamp across all messages in a run)	ms	Client side, per message
Throughput	Total messages received by all clients / run duration (10,000 ms)	msg/s	Aggregated at run end
CPU utilization	currentLoad from systeminformation, averaged over client-spawn samples	%	Testing container OS
Memory usage	Mem used / 1024^2 from systeminformation	MB	Test container OS

Table 2: Metric definitions, units and measurements units.

VI. RESULTS AND DISCUSSION

The following sections walk through each metric in turn. Some patterns are consistent across all four; others are more nuanced. Where the data points in one direction but the interpretation is uncertain, we try to say so rather than projecting more confidence than the single-run design supports.

A. Message Latency

The data in Table 3 confirms the structural argument laid out above. At 10 concurrent clients, REST records the lowest latency (1.67 ms), while WebSocket (3.28 ms) and SSE (3.56 ms) both carry a small overhead from their respective connection-setup paths. This advantage reverses sharply from 50 users onward: REST latency climbs to 9.57 ms at 50 users, 16.78 ms at 100, and plateaus around 20–22 ms between 500 and 1,000 users, indicating event-loop saturation. WebSocket latency, by contrast, actually *falls* from 3.28 ms at 10 users to a remarkably stable 0.53–0.60 ms band from 100 to 1,000 users — consistent with a connection whose per-message work is near-constant regardless of concurrent connection count. SSE follows a similar downward trend, reaching 0.70 ms at 500 users, before showing elevated variance (mean 9.75 ms, std 49.06 ms) at 1,000 users. That high standard deviation at the 1,000-user mark signals occasional scheduling spikes rather than a sustained latency increase; the median is likely closer to the 500-user figure, but the tail events pull the mean upward. Across all five concurrency levels, WebSocket is the most consistently low-latency and lowest-variance protocol.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

Concurrent Users	REST	WebSocket	SSE
10	1.67 ± 0.35	3.28 ± 6.21	3.56 ± 3.11
50	9.57 ± 1.63	0.99 ± 0.90	1.49 ± 1.15
100	16.78 ± 4.08	0.57 ± 0.16	0.88 ± 0.28
500	20.86 ± 2.24	0.53 ± 0.20	0.70 ± 0.28
1000	22.40 ± 3.21	0.60 ± 0.39	9.75 ± 49.06

Table 3: Average latency in milliseconds

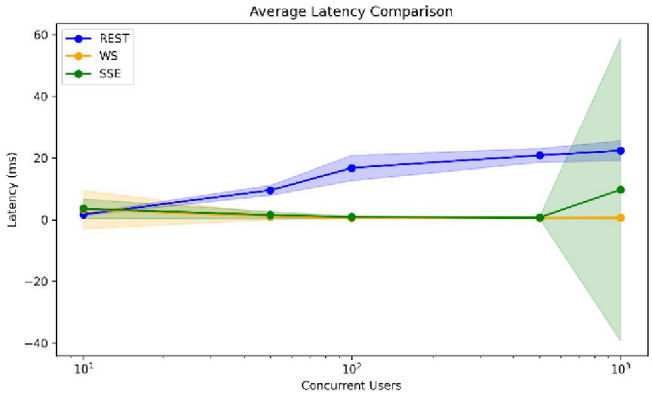

Fig 2. Line Chart – Average Latency vs Concurrent Users (REST, SSE and WS)

B. Throughput

Table 4 shows throughput figures that require careful interpretation because the measurement semantics differ between REST and the push-based protocols. WebSocket and SSE both deliver approximately one message per second per client — 0.01 msg/s at 10 users, scaling linearly to 0.90 msg/s at 1,000 users — because their servers push on a fixed one-second interval regardless of client request behaviour. REST throughput, measured as completed poll cycles per second across all clients, sits at 4.73 msg/s at 10 users and rises to 44.85 msg/s at 1,000 users. The higher absolute number for REST is an artefact of its pull model: each sequential polling loop can complete many request-response cycles per second when round-trip times are short (as they are inside Docker), whereas push-based servers are self-throttled to one event per second per client. In a real deployment where update frequency matters — say, 10 updates per second — REST's per-request overhead would compound, while WebSocket and SSE need only adjust the server-side interval. The near-zero standard deviation for WebSocket and SSE throughput across all concurrency levels reflects the deterministic nature of the timer-driven push: message delivery is consistent run-to-run. REST's comparatively higher standard deviation (e.g., ±4.92 at 1,000 users) reflects variation in event-loop service time as request queues grow.

Concurrent Users	REST	WebSocket	SSE
10	4.73 ± 1.02	0.01 ± 0.00	0.01 ± 0.00
50	4.86 ± 0.76	0.05 ± 0.00	0.05 ± 0.00
100	6.31 ± 1.12	0.09 ± 0.00	0.09 ± 0.00
500	24.34 ± 2.47	0.45 ± 0.00	0.45 ± 0.00
1000	44.85 ± 4.92	0.90 ± 0.00	0.90 ± 0.02

Table 4: Aggregate throughput(msg/s)

DOI: 10.48175/IJARSCT-35586

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

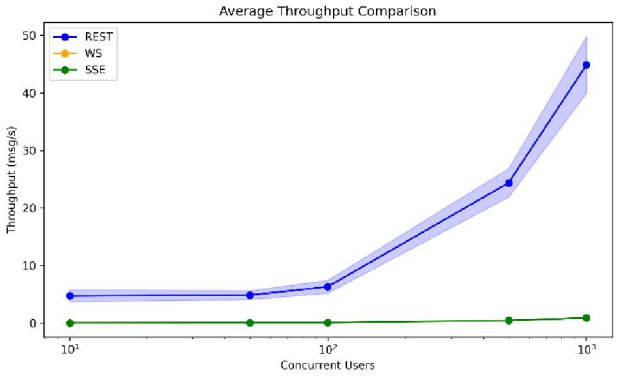

Fig 3: Line Chart – Throughput by Protocol at Each Concurrency Level

C. CPU Utilisation

The CPU figures in Table 5 are the most striking in the dataset. REST consumes roughly 43–50% CPU across all concurrency levels, with the utilisation essentially flat from 10 to 1,000 users. This plateau suggests that even at 10 users, the REST server's polling loop is driving the event loop close to its steady-state processing rate for this workload — adding more users increases queue depth but not peak CPU, because the loop is already running hot. WebSocket and SSE both start very low (approximately 2.5–4% at 10–100 users) and rise steeply at 500 and 1,000 users as the per-client timer callbacks accumulate. At 1,000 users, WebSocket reaches 40.1% and SSE reaches 36.5% — both meaningfully below REST's 49.9%. SSE's lower CPU figure at every concurrency level above 100 users supports the hypothesis that res.write() is computationally cheaper per message than a ws.send() wrapped in a WebSocket frame, consistent with SSE's simpler wire format. The implication for large fan-out deployments (e.g., push notifications to tens of thousands of subscribers) is that an SSE-based architecture may reach CPU saturation later than a WebSocket-based one.

Concurrent Users	REST	WebSocket	SSE
10	43.58 ± 1.47	2.55 ± 1.02	2.66 ± 0.98
50	44.27 ± 1.12	2.93 ± 0.92	2.52 ± 0.71
100	46.59 ± 1.89	4.00 ± 2.93	3.15 ± 1.02
500	50.08 ± 0.57	32.74 ± 2.45	26.90 ± 3.55
1000	49.92 ± 0.93	40.11 ± 2.42	36.51 ± 2.55

Table 5: Average CPU utilisation (%)

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

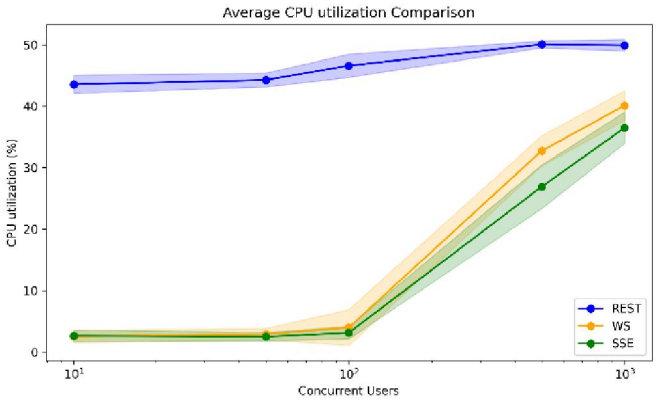

Fig 4: Line chart – CPU Utilisation vs. Concurrent Users

D. Memory Usage

Memory usage across all three protocols and concurrency levels is presented in Table 6. The range across all conditions is narrow — approximately 2,065 MB to 2,230 MB — indicating that the dominant memory consumer in the test environment is the testing container itself (Node.js runtime, loaded modules, and client-side state), not the per-connection overhead of any individual protocol. Nonetheless, relative trends are visible. REST memory is consistently the highest at each user level from 10 to 500 users, which is somewhat counter-intuitive given REST's stateless design, but likely reflects the memory footprint of concurrent in-flight HTTP request objects held in Express.js middleware. SSE consistently shows the lowest memory at 10 through 500 users, with REST and SSE converging at 1,000 users (2,229.63 MB and 2,228.93 MB respectively). WebSocket sits between the two at all levels. The linear growth from 10 to 1,000 users is consistent across all three protocols (approximately 60–165 MB total growth), confirming that connection-state memory scales predictably. The convergence of all three protocols near 2,200–2,230 MB at 1,000 users suggests that at this concurrency level, runtime overhead dominates over protocol-specific per-connection cost.

Concurrent Users	REST	WebSocket	SSE
10	2166.97 ± 40.08	2114.77 ± 34.60	2065.04 ± 27.18
50	2153.85 ± 32.79	2126.98 ± 37.45	2075.39 ± 32.04
100	2164.89 ± 30.64	2152.71 ± 38.89	2079.33 ± 31.46
500	2195.79 ± 31.93	2171.38 ± 40.12	2142.05 ± 29.94
1000	2229.63 ± 26.02	2206.88 ± 31.67	2228.93 ± 32.89

Table 6: Average Memory usage (MB)

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

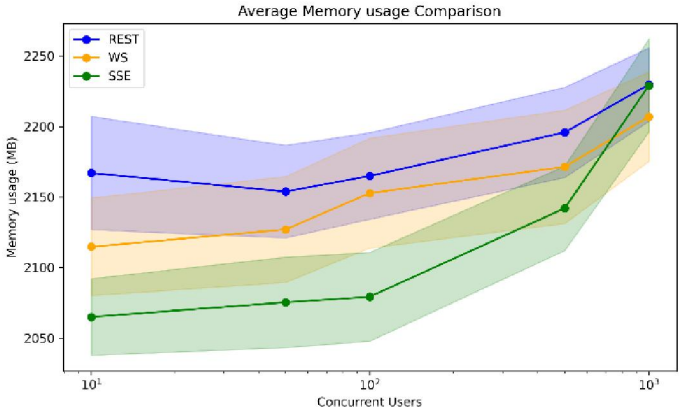

Fig 5: Line Chart – Memory Usage by Protocol and Concurrency Level

E. Synthesis

Stepping back from the per-metric analysis, the four metrics converge on a coherent picture, summarised in Table 7. WebSocket is the strongest choice for applications that require bidirectional, low-latency, high-frequency messaging — the latency figures from 50 users onward are an order of magnitude below REST, and throughput is stable and predictable. That capability comes with per-client connection state, moderate CPU growth under high concurrency, and infrastructure requirements that do not apply to plain HTTP. SSE is a genuinely competitive option for unidirectional server-push workloads: it consumed less CPU than either alternative at high concurrency, operates through all standard HTTP proxies and CDNs without special configuration, and provides built-in reconnection that WebSocket applications must implement themselves. REST remains entirely appropriate — and incurs no meaningful penalty — for APIs where data access is request-driven or infrequent. Its costs only become visible when it is pressed into service as a near-real-time delivery mechanism through high-frequency polling, at which point its per-request event-loop overhead accumulates into measurable latency and sustained high CPU utilisation.

Metric	REST	WebSocket	SSE
Latency at low load (10 users)	Best (1.67 ms) Comparable (3.28 ms) Comparable (3.56 ms)	Best (1.67 ms) Comparable (3.28 ms) Comparable (3.56 ms)	Best (1.67 ms) Comparable (3.28 ms) Comparable (3.56 ms)
Latency at high load (1000 users)	Worst (22.40 ms)	Best (0.60 ms)	Good (9.75 ms avg, high variance)
Throughput scaling	Strong (scales with polling rate)	Proportional to push interval	Proportional to push interval
CPU at high load (1000 users)	Highest (~50%)	Moderate (~40%)	Lowest (~37%)
Memory at high load (1000 users)	Highest (2229 MB)	Moderate (2207 MB)	Comparable (2229 MB)
Infrastructure compatibility	Universal	Requires proxy support	Universal (plain HTTP)
Reconnection handling	Manual	Manual	Built-in (EventSource)

Table 7: Summary of protocol performance characteristics across all four metrics

DOI: 10.48175/IJARSCT-35586

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

VII. THREATS TO VALIDITY

We want to be honest about what these results can and cannot support, because a benchmark that does not acknowledge its limitations tends to get cited beyond the scope it was designed for.

1. Network environment. Docker's internal bridge network has sub-millisecond RTTs and gigabit-equivalent throughput between containers on the same host. Real applications operate over LANs, WANs, or the public internet, all of which add latency and reduce bandwidth. The relative ordering of protocols is unlikely to change, but the magnitude of differences — particularly between REST's per-request overhead and the persistent-connection protocols — will be larger in practice.

2. Message frequency. All three servers push or respond with one message per second per client. This is a low message rate. Applications with sub-100ms update requirements — high-frequency trading, real-time gaming, live telemetry — will amplify the per-message overhead differences. At 10 messages per second, the REST event loop load is ten times what we measured; the WebSocket and SSE overhead scales less steeply because their per-message cost is lower.

3. Single-run design. Each protocol-concurrency combination was measured once. We have no characterisation of run-to-run variance. Node.js garbage collection, OS scheduler pre-emption, and Docker network stack fluctuations all introduce noise that a single measurement cannot distinguish from signal. Results should be treated as indicative rather than statistically definitive.

4. No JIT warm-up. V8 starts in interpreted mode and progressively compiles hot code paths. The first requests or messages in a run incur compilation overhead that later ones do not. A warm-up phase of at least 5–10 seconds before measurement begins would reduce this effect.

5. Payload size. The JSON messages are small — 40 to 60 bytes depending on counter value. REST's per-request HTTP header overhead is comparable in size to the payload, which exaggerates its relative cost. For payloads in the kilobyte range, the header overhead becomes proportionally smaller, and REST's position relative to the other two would improve.

6. Single runtime and language. Node.js's event-loop concurrency model is well-matched to I/O-bound persistent connections. A thread-per-connection server in Java or Go would distribute connection state differently and might show REST performing better relative to the persistent-connection protocols than our results suggest.

VIII. CONCLUSION

We set out to answer a straightforward question — which of REST, WebSocket, and SSE performs better, and under what conditions — with a reproducible, measurement-based answer rather than another round of qualitative discussion. The benchmarking tool we built makes that question answerable for any developer willing to clone the repository and run docker-compose up. Thirty experimental runs across five concurrency levels produced a dataset with enough consistency to draw conclusions with reasonable confidence, while the standard deviations honestly surface where variability remains.

The results broadly confirm what experienced practitioners tend to believe but rarely have concrete numbers for, and they add some detail that is not obvious from first principles. WebSocket delivered the lowest and most stable latency from 50 users onward, reaching 0.60 ms at 1,000 concurrent clients — roughly 37 times lower than REST at the same load. Its throughput was perfectly consistent across all runs. These figures make it the unambiguous choice for

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 8, May 2026 Impact Factor: 8.2

applications that require bidirectional, low-latency, high-frequency communication: interactive games, collaborative editors, trading interfaces, and chat systems where both client and server need to initiate messages.

SSE performed better than its reputation might suggest. It matched WebSocket's throughput precisely at every concurrency level, consumed less CPU than either competitor under high load — 36.51% versus 40.11% for WebSocket and 49.92% for REST at 1,000 users — and showed the lowest memory footprint at low to moderate concurrency. The elevated latency variance at 1,000 users is a limitation worth watching, but for the large class of server-push workloads — dashboards, live feeds, notification systems, telemetry displays — where the client never needs to send data back to the server, SSE provides a compelling combination of performance, infrastructure simplicity, and built-in reliability that WebSocket cannot match without additional application-level code. Developers who default to WebSocket for every real-time requirement should consider whether SSE's simpler operational profile is a better fit for their actual data flow.

REST's position in these results is not a demotion. Its costs — sustained ~50% CPU utilisation and latencies climbing toward 22 ms under 1,000-user polling load — are the direct consequence of applying a request-response protocol to a continuous-stream workload it was not designed for. For the vast majority of web API interactions — resource retrieval, form submission, CRUD operations, infrequent data access — those costs never materialise, and REST's advantages in cache ability, tooling compatibility, and operational familiarity are unreservedly valuable. The data here is a caution against high-frequency polling, not against REST itself.

The testing tool is perhaps the most reusable contribution of this work. By packaging everything in Docker Compose with a single-entry point, the experiment is accessible to any developer without requiring a specific operating system, language runtime, or hardware configuration. The tool can be extended to additional protocols — gRPC-Web, Web Transport, and MQTT over WebSocket are natural candidates — by adding a new server container and client module without modifying the measurement harness or the CSV output schema.

Future work should address the study's main limitations: multiple runs with reported variance, a proper warm-up phase to allow V8 JIT compilation to stabilise, testing over a realistic wide-area network to amplify the per-request overhead differences that Docker's internal network compresses, varying payload sizes to identify the crossover point at which REST's header-to-payload ratio improves enough to change the relative standings, and extending the concurrency range beyond 1,000 to locate the saturation point of each protocol. Automated chart generation from the CSV output would also lower the barrier to re-running and comparing modified configurations.

REFERENCES

1. S. Loreto, S. P. Romano, M. Totaro, and G. Wilkins, "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP," IETF RFC 6202, April 2011.
2. V. Pimentel and B. G. Nickerson, "Communicating and Displaying Real-Time Data with WebSocket," IEEE Internet Computing, vol. 16, no. 4, pp. 45–53, July–Aug. 2012.
3. I. Fette and A. Melnikov, "The WebSocket Protocol," IETF RFC 6455, December 2011.
4. D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," IEEE Software, vol. 35, no. 3, pp. 56–62, May/June 2018.
5. L. Wang, E. Bao, J. Lu, and M. Zhao, "Comparison of REST and gRPC in Microservice Applications," in Proc. 2nd Int. Conf. on Big Data and Artificial Intelligence (BDAI), pp. 58–63, 2019.
6. R. Souza, C. Uchôa, and M. Monteiro, "Server-Sent Events vs. WebSockets vs. HTTP Polling: A Micro-Benchmark Study in Browser Environments," in Proc. IEEE/ACM SBMF, pp. 45–52, 2021.
7. A. Chaniotis, K.-I. D. Kyriakou, and N. D. Tselikas, "Is Node.js a Viable Option for Building Modern Web Applications? A Performance Evaluation Study," Computing, vol. 97, no. 10, pp. 1023–1044, 2015.
8. T. H. Noor, S. Zeadally, A. Alfazi, and Q. Z. Sheng, "Mobile Cloud Computing: Challenges and Future Research Directions," Journal of Network and Computer Applications, vol. 115, pp. 70–85, 2018.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

9. R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.
10. WHATWG, "Server-Sent Events," HTML Living Standard. [Online]. Available: https://html.spec.whatwg.org/multipage/server-sent-events.html