International Journal of Computer (IJC)
ISSN 2307-4523 (Print & Online)
https://ijcjournal.org/index.php/InternationalJournalOfComputer/index

SSE Event-Contract Design for Frontend Communication with Intelligent Agents: An Analytical Architectural Perspective

Tounzal Elias*

Software engineer at Quaestor Technologies, Inc, San Francisco, US
Email:elias.tounzal@gmail.com

Abstract

This article develops an analytical architectural perspective on the use of Server-Sent Events (SSE) in frontend communication with intelligent agents in user-facing systems. The study synthesizes a bounded corpus of ten publications spanning real-time web transport, agent communication protocols, multi-agent orchestration, dynamic interface generation, observability, retrieval-grounded generation, output-layer security, and lifecycle evaluation. The purpose is to formulate a transparent conceptual model of browser-facing event streaming and to derive architecture-level design propositions for stable rendering, controlled state exposure, observable execution, governed latency handling, and tenant-safe delivery. The analysis indicates that SSE is analytically well-suited to browser-facing agent streaming when implemented as a typed event channel within a broader event-governed architecture. The article should therefore be read as a conceptual architectural proposal for React-based interfaces, agent platforms, and production teams designing streaming assistants.

Keywords: server-sent events; intelligent agents; frontend architecture; React; streaming protocols; agent communication; generative UI; observability; RAG; output validation.

Received: *2/10/2026*
Accepted: *4/10/2026*
Published: *4/17/2026*

---

* *Corresponding author.*

1. Introduction

The shift from static request–response interfaces to agent-mediated interaction has altered the technical requirements imposed on frontend communication. In conventional web applications, the browser usually receives a completed payload and renders a stable view. In agent-based systems, the browser increasingly receives an evolving sequence of intermediate states: partial text, tool-calling intentions, retrieved evidence, execution progress, error states, cancellation acknowledgements, and structured interface directives. Such interaction patterns place pressure on transport design, as the communication layer now participates in user experience, reliability controls, and security boundaries.

Within this setting, SSE has become a pragmatic transport option for assistant output streams because it preserves the HTTP model and supports incremental delivery to the client while keeping the frontend implementation comparatively transparent. Yet a plain adoption of SSE does not resolve the more complex problem. When intelligent agents rely on multi-step orchestration, sub-agents, retrieval components, and external tools, the stream can easily degenerate into an unstable mixture of raw text fragments and backend-specific events. The result is a brittle interface, complex cancellation semantics, weak observability, and increased exposure to cross-session leakage or malformed component rendering.

The purpose of the study is to formulate an architectural perspective on frontend communication with intelligent agents in which SSE is examined as the browser-facing streaming layer within a broader event-governed system. The study addresses three tasks. First, it identifies the architectural properties that make SSE suitable for browser-facing agent streaming and defines the limits of that suitability. Second, it explains how event contracts connect transport, rendering, and orchestration, and provide visibility and control semantics in a React environment. Third, it derives architecture-level design propositions linking the streaming layer to observability, grounding, evaluation, and output safety.

The study addresses three tasks. First, it identifies the architectural properties that make SSE suitable for frontend–agent communication and defines the limits of that suitability. Second, it establishes how the event model should be structured so that markdown responses and interactive components can be rendered in a controlled React environment. Third, it formulates optimization principles that connect transport design with orchestration, observability, evaluation, knowledge grounding, and security controls.

The article's contribution lies in integrating several recent research lines into a single architectural reading of frontend-agent streaming systems, treating SSE as a governed transport component within that model.

2. Materials and Methods

This article follows a targeted analytical-review design. The source corpus consists of the ten publications listed in References [1–10], published between 2021 and 2026. Source selection was guided by direct relevance to the stated research problem: browser-facing streaming between frontend applications and intelligent agents.

The corpus was assembled to cover the main design layers that shape such systems: engineering difficulties in

agent development [1], retrieval-grounded generation [2], observability [3], dynamic interface generation [4], agent communication protocols [5], multi-agent workflow design [6], architecture evaluation [7], real-time web transport [8], output-layer security [9], and lifecycle-oriented evaluation [10]. A source was retained when it satisfied two conditions: first, it addressed a structural property directly related to frontend-agent streaming; second, it contributed a design rule, risk pattern, or quality attribute that could be compared across publications. Publications discussing large language models in more general terms, without direct implications for transport semantics, frontend state handling, protocol boundaries, observability, or governed execution, were not included in the analytical corpus.

The synthesis proceeded in four steps. First, each source was coded by analytical function: transport, orchestration, rendering, observability, security, retrieval grounding, or evaluation. Second, recurrent claims were extracted and normalized into comparable design statements, including event typing, state exposure, validation boundaries, trace identifiers, and user-visible grounding signals. Third, these statements were compared across the corpus to identify convergence, complementarity, and scope limits. Fourth, the convergent statements were assembled into a layered architectural model of SSE-based frontend-agent streaming.

The article does not claim exhaustive coverage of the literature or empirical validation. Its methodological contribution lies in a transparent analytical synthesis of a bounded source corpus into a reproducible architectural proposal.

3. Results

A productive discussion of SSE in agent interfaces has to begin with a distinction that is often blurred in implementation practice. SSE is a transport envelope for server-to-client streaming; it does not itself define how the stream represents lifecycle states, tool invocations, partial outputs, or interface instructions. Many implementation failures attributed to streaming therefore arise from weak event semantics. Studies on real-time web communication show that SSE belongs to the family of long-lived HTTP-based mechanisms developed for continuous browser updates [8]. In contrast, recent protocol surveys indicate that modern agent-facing communication increasingly depends on structured event streams that connect backend execution with user-facing applications [5].

For frontend integration, the stream must be treated as a typed event log. A token-only stream is sufficient for a minimal chat window, but it becomes structurally inadequate when the backend performs retrievals, switches among sub-agents, or emits actionable interface state. The communication layer, therefore, requires at least four semantic strata: session lifecycle events, content events, orchestration events, and control events. Session lifecycle events establish stream initiation, heartbeat, graceful termination, and recovery points. Content events carry textual deltas, citations, and finalized message blocks. Orchestration events surface tool requests, tool completion, agent handoffs, and retrieval checkpoints. Control events handle cancellation, retry, client acknowledgement, and recoverable errors. This layered view aligns with current agent-protocol research, which frames streaming communication as a sequence of structured events [5]. It also answers the engineering concerns observed in developer discussions, where interaction contracts and runtime robustness emerge as recurring sources of

implementation friction [1].

Once the event contract is typed, the frontend can separate parsing from rendering. That separation is decisive in React-based systems that need to display markdown while supporting interactive components. A typical implementation mistake is to let the same text channel carry prose, component instructions, and unstable fragments of backend metadata. Under streaming conditions, such blending produces malformed markdown, race conditions in reconciliation, and unsafe attempts to interpret model output as executable UI. Research on dynamic GUI generation in chat interfaces suggests a different approach: language-model output should guide interface composition via structured intermediaries, while the rendering layer remains under application control [4]. Applied to SSE, this implies that markdown text should be accumulated through content events, normalized into stable render boundaries, and passed to a trusted markdown renderer. At the same time, the interface widgets should be instantiated only from validated structured payloads mapped to predefined React components. The stream feeds the UI state machine. Figure 1 presents the original synthesized model used in this article. It was constructed by integrating protocol-layer observations from [5], interface-control principles from [4], orchestration abstractions from [6], and architecture/evaluation requirements from [7, 10].

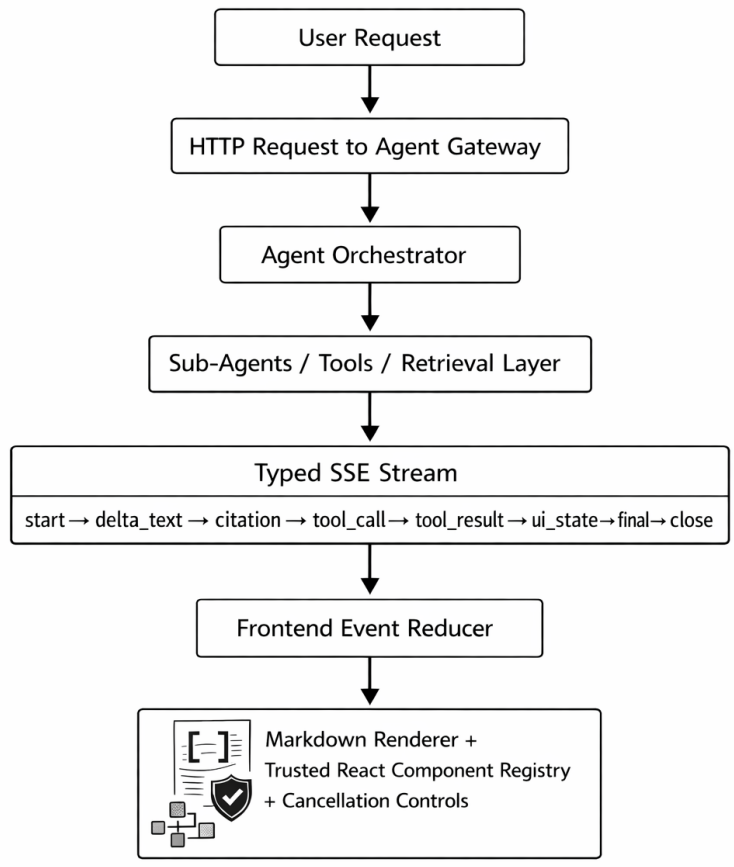

Figure 1: Synthesized event-layer model for SSE-based communication between frontend applications and intelligent agents, derived from [4–7, 10]

In this configuration, the frontend ceases to be a passive receiver of text and becomes a reducer of typed execution state. That shift yields two immediate gains. First, cancellation acquires a precise meaning. The client no longer

"stops the text"; it requests termination of an identified run or execution branch, and the server acknowledges the resulting state transition. Second, resumability becomes technically tractable because the client can reconstruct the visible state from replayable event categories. Such discipline is consistent with current work on agent architecture and evaluation, where architectural decisions are assessed through their effects on quality attributes under conditions of non-determinism and continuous evolution [7, 10].

The organization of backend orchestration directly influences stream design. Multi-agent systems distribute work across modules for perception, reasoning, action, memory, and inter-agent exchange [6]. Yet exposing that internal complexity verbatim to the frontend would over-couple the client to the backend implementation. The stream, therefore, needs a selective disclosure policy. The browser requires the subset of execution state that shapes user experience: which stage is active, whether a tool is pending, which retrieved evidence has become user-visible, whether a response is provisional, and whether human confirmation is required. It does not need raw internal chatter among sub-agents. Recent studies on agent communication emphasize protocol fragmentation as a practical obstacle; frontend-facing communication is easier to stabilize when the backend collapses internal heterogeneity into a small, versioned vocabulary of externally meaningful events [5], while keeping orchestration logic behind the gateway [6].

Latency in this study is treated as an architectural property. The reviewed literature does not provide controlled comparative measurements for the specific frontend-agent configuration examined here. For that reason, the discussion focuses on three analytical latency layers: transport establishment, time-to-first-meaningful-event, and time-to-stable-render. From that perspective, semantically earlier and cleaner event exposure appears more consequential for perceived responsiveness than indiscriminate token flushing [8]. The present article proposes event shaping and frontend buffering as architecture-level priorities for future implementation and testing.

The same architecture must remain observable. Once a frontend receives partial text, component instructions, and orchestration state over a long-lived stream, debugging through ordinary request logs becomes insufficient. Observability research in LLM systems argues for lifecycle-wide tracing that links high-level behavior to runtime evidence [3], while evaluation-driven agent engineering places monitoring inside a feedback loop that informs redesign and redeployment [10]. In an SSE-based interface, this requires every stream to carry stable identifiers for session, run, user scope, tool invocation, and event sequence. On the frontend side, these identifiers support replay, trace correlation, and defect localization. On the backend, they enable determining whether a failure originated in retrieval, orchestration, model behavior, or presentation mapping. The stream contract, therefore, doubles as an observability schema.Security constraints sharpen the same conclusion. User-facing agent systems process sensitive prompts, tenant-scoped knowledge, tool outputs, and intermediate reasoning traces. A streaming interface widens the attack surface if the server forwards unvalidated content directly into the render pipeline or leaks residual state across sessions. Output-layer security research argues that validation at the response boundary is a necessary complement to input filtering, especially for detecting prompt injection, sensitive data exposure, and policy-violating outputs [9]. Under an SSE model, such validation should occur before the event enters the outward stream. This means that text deltas, citations, tool outputs, and UI directives require separate validation rules. Plain prose and component metadata should never share the same trust level. The frontend should therefore consume a stream that has already crossed a validation boundary and restrict component rendering to a closed

registry of allowed interfaces [4, 9].

A final optimization question concerns precision. For many production assistants, the most unstable layer is the knowledge source from which answers are assembled. Research on retrieval-augmented generation shows that grounding model output in external evidence remains attractive when the knowledge base changes over time and when traceable source linkage is desirable [2]. For frontend-agent communication, the implication is straightforward: SSE should stream evidence-aware events. Retrieval completion, evidence selection, and citation materialization are part of the interaction contract because they shape what the user sees and what they trust. Fine-tuning remains useful for stable behavioral tendencies, formatting discipline, and domain-specific response style, but volatile operational knowledge is better surfaced through retrieval-aware streams [2]. When these streams are evaluated through architecture-sensitive and lifecycle-oriented procedures, SSE becomes one component of a broader control system.

4. Discussion

The analytical results indicate that the main design problem does not lie in choosing SSE over competing protocols. It lies in deciding what the stream is allowed to mean. When SSE is treated as an event ledger with a controlled semantic vocabulary, it aligns well with the requirements of user-facing intelligent agents. When reduced to token delivery, the frontend inherits the backend's ambiguity and becomes fragile under tool use, sub-agent orchestration, and interactive rendering. Table 1 systematizes the differences in transport patterns at the level most relevant to this article.

Table 1: Transport patterns for frontend communication with intelligent agents: analytical comparison derived from [1, 5, 8, 10]

Criterion	**Polling / repeated HTTP**	**SSE**	**WebSocket**
Browser communication model	Discrete request cycles	Long-lived server-to-client HTTP stream	Full-duplex persistent channel
Suitability for incremental assistant output	Low	High	High
Contract simplicity for frontend teams	Moderate	High	Lower at the browser boundary
Natural fit for one-way textual and state streaming	Weak	Strong	Strong, though often underused for purely outbound streams
Ease of aligning with HTTP infrastructure, auth, and logs	High	High	More variable
Risk of event-schema neglect	Moderate	High if implemented as a raw token flow	High if used without strict message semantics
Value for rich bidirectional tool/session control	Limited	Moderate, usually complemented by separate control endpoints	High

Note. The comparative descriptors in Table 1 were assigned through rule-based cross-source synthesis. "High" denotes repeated direct support in the cited corpus for browser-facing assistant streaming under the stated criterion; "Moderate" denotes partial or conditional support; "Low" denotes limited direct support. Browser communication model and infrastructure alignment are derived primarily from [8]; contract simplicity and event-schema risk from [1] and [5]; lifecycle and session-governance considerations from [10].

The comparison shows why SSE has become attractive in frontend-heavy agent products. It occupies a middle position: sufficiently expressive for incremental delivery, yet less operationally intensive than a fully bidirectional socket layer when the dominant traffic direction is outbound from the agent to the browser. At the same time, the table makes clear that transport choice cannot substitute for protocol design. A poor event model can damage reliability under any transport.

The second implication concerns interface rendering. Recent work on generative interfaces and agent communication suggests that chat UIs are drifting away from plain transcript display toward stateful interaction

surfaces [4, 5]. That tendency is especially relevant for React applications, where a single assistant response may include narrative explanations, citations, data views, and interaction affordances. Table 2 translates this shift into architectural priorities.

Table 2: Architecture-level priorities for frontend-agent streaming with SSE as the browser-facing transport layer: analytical synthesis derived from [2–7, 9, 10]

Optimization priority	**Architectural implication**	**Frontend consequence**	**Backend consequence**
Stable rendering of markdown and widgets	Separate text events from UI-state events	Trusted markdown pipeline and closed component registry	Structured event emission instead of mixed free-form payloads
Runtime observability	Attach identifiers to session, run, tool, and event sequence	Easier replay and defect localization	Trace correlation across model, tool, and stream layers
Multi-agent orchestration transparency	Expose user-relevant stage transitions, conceal internal chatter	Clear progress states without backend over-coupling	Gateway-level normalization of heterogeneous internal workflows
Precision under changing knowledge	Surface retrieval-aware states and source materialization	Users can see grounded output formation	Retrieval pipeline integrated into the event contract
Safety and tenant isolation	Validate output before emission; isolate session memory	Lower risk of unsafe rendering or cross-user bleed	Output-layer policy enforcement and scoped memory handling
Controlled evaluation	Define measurable stream events and terminal states	UX metrics linked to stream milestones	Offline and online evaluation built around event traces

Note. Table 2 summarizes synthesis outputs. Each row was derived by linking a recurrent frontend requirement to a recurrent backend design response in the reviewed corpus: stable rendering [4], observability [3, 10], orchestration transparency [5, 6], knowledge grounding [2], safety and tenant isolation [4, 9], and controlled evaluation [7, 10].

The interpretive value of this synthesis lies in how it brings together concerns that are often separated

organizationally. Frontend teams tend to focus on rendering continuity and cancellation. Backend teams tend to focus on orchestration and model quality. Security teams emphasize leak prevention. Platform teams prioritize logging and evaluation. In agent systems, these concerns intersect at the stream boundary. An SSE design that omits structured identifiers, validation stages, or retrieval-aware milestones simultaneously weakens all four domains.

The present article does not report prototype measurements, latency benchmarks, controlled comparisons across transport layers, or user-level experiments. Its contribution lies in clarifying a reproducible synthesis logic for frontend-agent streaming design. The propositions offered here should therefore be read as analytically derived design hypotheses and architecture guidelines that invite later prototyping and empirical evaluation.

For frontend web development, the consequences are clear. A robust assistant UI in React should be built around an event reducer, a trusted markdown layer, a versioned schema for orchestration and retrieval states, and a narrow component registry. For intelligent-agent engineering, the corresponding consequence is equally clear: the backend should export a frontend-facing protocol that summarizes execution in stable event categories while keeping internal multi-agent complexity encapsulated. Under these conditions, SSE remains a strong architectural choice for user-facing intelligent systems.

5. Conclusion

The first task of the study was to identify the architectural conditions under which SSE can serve as a browser-facing transport for intelligent-agent systems. The analysis suggests that SSE is analytically suitable where outbound incremental updates dominate and where the control plane remains narrow, explicit, and externally stable.

The second task was to clarify how an event contract can support markdown rendering and interactive components in React. The analysis indicates that stable frontend integration depends on separating textual content, orchestration state, control signals, and validated UI directives into distinct event classes.

The third task was to derive architecture-level design propositions linking streaming with observability, grounding, evaluation, and security. Within the limits of the reviewed literature, the article argues for treating SSE as a transport layer within a broader frontend-agent streaming architecture. The proposed model remains analytical and awaits prototype- or benchmark-based validation.

References

[1] Asgari, A., Panichella, A., Derakhshanfar, P., & Olsthoorn, M. (2025). What Challenges Do Developers Face in AI Agent Systems? An Empirical Study on Stack Overflow. CoRR, abs/2510.25423. https://doi.org/10.48550/arXiv.2510.25423

[2] Brown, A., Roman, M., & Devereux, B. (2025). A systematic literature review of retrieval-augmented generation: Techniques, metrics, and challenges. *Big Data and Cognitive Computing, 9*(12), 320. https://doi.org/10.3390/bdcc9120320

[3] Chen, X., Li, Y., & Wang, X. (2025). Design Principles and Guidelines for LLM Observability: Insights from Developers. In Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA '25), 177:1–177:9. https://doi.org/10.1145/3706599.3719914

[4] Hojo, N., Shinoda, K., Yamazaki, Y., Suzuki, K., Sugiyama, H., Nishida, K., & Saito, K. (2025). GenerativeGUI: Dynamic GUI generation leveraging LLMs for enhanced user interaction on chat interfaces. In *Extended abstracts of the CHI conference on human factors in computing systems (CHI EA '25)* (Article 306, pp. 1–9). Association for Computing Machinery. https://doi.org/10.1145/3706599.3719743

[5] Kong, D., Lin, S., Xu, Z., Wang, Z., Li, M., Li, Y., Zhang, Y., Sha, Z., Li, Y., Lin, C., Wang, X., Liu, X., Khan, K., Zhang, N., Chen, C., & Han, M. (2025). A survey of LLM-driven AI agent communication: Protocols, security risks, and defense countermeasures. *arXiv*. https://doi.org/10.48550/arXiv.2506.19676

[6] Li, X., Wang, S., Zeng, S., et al. (2024). A survey on LLM-based multi-agent systems: Workflow, infrastructure, and challenges. *Vicinagearth*, *1*, 9. https://doi.org/10.1007/s44336-024-00009-2

[7] Lu, Q., Zhao, D., Liu, Y., Zhang, H., Zhu, L., Xu, X., Shi, A., Tan, T., & Kazman, R. (2026). AgentArcEval: An architecture evaluation method for foundation model based agents. Journal of Systems and Software, 232, 112656. https://doi.org/10.1016/j.jss.2025.112656

[8] Murley, P., Ma, Z., Mason, J., Bailey, M., & Kharraz, A. (2021). WebSocket adoption and the landscape of the real-time web. In *Proceedings of the Web Conference 2021* (pp. 1192–1203). https://doi.org/10.1145/3442381.3450063

[9] Podpora, M., Baranowski, M., Chopcian, M., Kwasniewicz, L., & Radziewicz, W. (2026). LLM firewall using validator agent for prevention against prompt injection attacks. *Applied Sciences, 16*(1), 85. https://doi.org/10.3390/app16010085

[10] Xia, B., Lu, Q., Zhu, L., Xing, Z., Zhao, D., & Zhang, H. (2024). Evaluation-driven development and operations of LLM agents: A process model and reference architecture. *arXiv*. https://arxiv.org/abs/2411.13768