Persistent WebSocket NLP Infrastructure as a Natural Transport Architecture for LLM-Based Autonomous Agents

Research Analysis — Perplexity Computer

March 2026

Abstract

Large language model (LLM)-based autonomous agents have emerged as a dominant paradigm for enterprise task automation, yet their deployment in production environments remains constrained by systemic inefficiencies in tool-calling infrastructure. This paper identifies and formalizes five critical architectural pain points in enterprise agent systems — connection overhead accumulation, stateless context loss, polling-induced latency, backpressure absence, and protocol-level incompatibility with emerging agent standards — and demonstrates that a persistent WebSocket connection serving a unified NLP microservices pipeline represents an architecturally well-aligned solution. We analyze the agent execution loop (reason-act-observe) and show that its iterative, stateful, event-driven nature is fundamentally misaligned with the request-response semantics of REST/HTTP APIs that dominate current NLP service offerings. Through analysis of the WebSocket protocol (RFC 6455), the Model Context Protocol (MCP) specification, and recent empirical findings on agent tool-calling overhead, we establish that a single persistent WebSocket connection carrying multiplexed NLP operations — including entity extraction, coreference resolution, cross-encoder reranking, topic modeling, causal analysis, and vectorization — eliminates per-call handshake overhead, enables connection-scoped session state, supports progressive result streaming, provides native backpressure flow control, and aligns with MCP's supported WebSocket transport mode. We further position this architecture as the tool-layer complement to context-window management approaches such as the demand-paging memory hierarchy for context windows[21], arguing that stateful NLP infrastructure and intelligent context eviction together describe a complete memory hierarchy for agentic AI. We estimate that for a typical enterprise agent executing 7–12 NLP tool calls per reasoning chain, this architecture analytically estimates a 35–45% reduction in cumulative tool-calling latency while simultaneously improving result quality through cross-operation context propagation.

1. INTRODUCTION

The deployment of LLM-based autonomous agents in enterprise environments has accelerated rapidly, with agents now orchestrating complex multi-step workflows across document analysis, business intelligence, customer support, and knowledge management domains. These agents operate through a fundamental execution loop: the LLM reasons about a task, selects and invokes external tools, observes the results, and iterates until the task is complete[1]. This loop — variously called the "reason-act-observe" cycle, the ReAct pattern, or simply the agent loop — is

the architectural primitive of all modern agent frameworks including LangChain, CrewAI, AutoGen, and Microsoft's Agent Framework.

Within this loop, Natural Language Processing (NLP) operations constitute the most frequently invoked tool category. An agent analyzing a business document will typically chain sentence segmentation, named entity recognition, coreference resolution, keyword extraction, topic modeling, vectorization, and cross-encoder reranking — often 7 to 12 distinct NLP operations per reasoning chain. Each operation produces outputs that inform subsequent operations: entities extracted in one step become inputs for coreference resolution in the next; resolved coreference chains improve the quality of topic modeling and causal analysis downstream.

Despite this inherently sequential, stateful, and streaming nature of NLP tool usage within agent loops, the overwhelming majority of commercial NLP APIs — including those from OpenAI, Google Cloud NLP, AWS Comprehend, Cohere, and Hugging Face — are built on stateless REST/HTTP architectures. While modern HTTP/2 deployments with persistent connection pools and TLS 1.3 session resumption mitigate some per-request overhead, the fundamental architectural mismatch — stateless request semantics serving stateful, interdependent NLP operations — persists regardless of transport optimization. In the common case of HTTP/1.1 deployments without connection pooling — still prevalent among commercial NLP APIs including AWS Comprehend, Google Cloud NLP, and Cohere — each API call initiates a new TCP connection, negotiates TLS, transmits the full request payload, waits for the complete response, and tears down the connection. For an agent executing 10 NLP operations per reasoning chain across 5 chains per task, this amounts to 50 independent connection lifecycles — each carrying 100–300ms of pure connection overhead before any NLP computation begins.

This paper argues that this architectural mismatch between the stateful, iterative, event-driven nature of agent execution and the stateless, request-response semantics of REST-based NLP services represents a fundamental bottleneck in enterprise agent deployment. We propose and analyze a WebSocket-native NLP architecture in which all NLP operations are served through a single persistent connection, and demonstrate its superiority across five critical dimensions: latency, state management, streaming capability, flow control, and protocol compatibility with emerging agent standards.

Concurrent work on context-window memory management[21] has independently identified the same statelessness problem from the context-window side. An empirical analysis of 857 production sessions encompassing 4.45 billion effective input tokens reveals 21.8% structural waste arising from stale tool results that persist in the LLM's context window, with dead tool output alone accounting for 26.5% of that waste. Where that work proposes a demand-paging memory hierarchy to manage waste after it enters the context window, the architecture presented here addresses the problem from the opposite end: a WebSocket-native NLP service prevents the waste from being generated in the first place by maintaining session state at the tool layer. Together, context-window management and stateful tool infrastructure describe complementary halves of a complete memory hierarchy for agentic AI.

2. BACKGROUND AND RELATED WORK

2.1 The Agent Execution Loop

Modern LLM-based agents operate through iterative tool-calling loops. Singh et al. describe this as an "LLM + loop + tools" architecture in which the agent's behavior emerges from the interaction between reasoning (the LLM), iteration (the loop), and external capabilities (the tools)[5]. The Routine framework formalizes this as a multi-step planning structure with explicit parameter passing between steps, achieving 96.3% execution accuracy on GPT-4o when the planning structure aligns with tool invocation patterns[2].

The latency characteristics of these loops have been studied extensively. Huang et al. demonstrate that agent execution is fundamentally serial: each step requires LLM reasoning followed by tool execution, and neither can proceed until the other completes[3]. Their SPAgent framework achieves 1.65x speedup by introducing speculative execution — predicting likely tool calls before reasoning completes — but the underlying serial dependency between reasoning and tool execution remains.

At the system level, Luo et al. identify that existing LLM serving systems treat each agent step as an independent request, failing to exploit the program-level structure of agent execution[4]. Their Autellix system achieves 4–15x throughput improvements by scheduling LLM calls based on their position within the agent program, demonstrating that awareness of the agent loop structure enables dramatic optimization.

2.2 Tool-Calling Overhead in Enterprise Agents

The overhead of tool calling in production agent systems has been documented from multiple angles. The Z-Space framework, deployed at Alibaba's Eleme platform, found that naive tool invocation consumed excessive tokens due to full-prompt injection of tool descriptions, reducing average token consumption by 96.26% through intelligent tool filtering[1]. Hathidara et al. report that near-duplicate tools in enterprise environments cause frequent invocation failures, with their DiaFORGE framework improving tool-invocation success by 27 percentage points over GPT-4o[6].

Xu et al. introduce the concept of "tool partial execution" in their Conveyor system, enabling tools to begin returning results before full computation completes, achieving up to 38.8% improvement in request completion latency[7]. This finding is particularly relevant to NLP pipelines, where progressive results (e.g., entities arriving before full coreference resolution completes) can feed the agent's reasoning earlier.

Singh et al. demonstrate that fusing parallel function calls into batched operations can dramatically reduce round-trip overhead, finding that compositional tool chains — where the output of one tool feeds the input of another — are the dominant pattern in production agent systems[5].

2.3 Communication Protocol Performance

The performance characteristics of WebSocket versus REST have been studied in both general web application and machine learning serving contexts. Badurowicz and Lasocha provide empirical measurements showing that WebSocket achieves significantly lower data transfer times than HTTP for repeated communications, with the advantage growing as the number of interactions increases[8].

Balu presents a comparative analysis of gRPC, REST, and WebSocket for streaming applications, finding that while gRPC achieves 2.5x higher throughput than REST, WebSocket remains the

preferred protocol for browser-based and edge clients where binary protocol support is limited[9]. Kaminski et al. benchmark REST, WebSocket, gRPC, GraphQL, and SOAP implementations in Python, providing detailed measurements of time and data transfer overhead across protocols[10].

Hassan provides a comprehensive framework for protocol selection in web applications, recommending WebSocket for "applications requiring real-time, bidirectional, persistent communication" and noting that "the persistent connection model of WebSocket eliminates per-request overhead that accumulates in high-frequency interaction patterns"[11].

2.4 RAG Pipeline Bottlenecks

The latency characteristics of Retrieval-Augmented Generation (RAG) systems — which share significant architectural overlap with NLP tool pipelines — provide relevant empirical context. Jiang et al. demonstrate that retrieval operations constitute a substantial portion of total generation time in RAG systems, and introduce PipeRAG to overlap retrieval and generation through pipeline parallelism, achieving 2.6x speedup[12]. Jin et al. identify the "long sequence due to knowledge injection" as the primary performance bottleneck in RAG systems and propose RAGCache to dynamically overlap retrieval and inference steps[13].

The Higress RAG system, built on the Model Context Protocol, demonstrates a "Full-Link Optimization" strategy that orchestrates adaptive routing, semantic caching, hybrid retrieval, and corrective RAG through a layered architecture, achieving 50ms-latency semantic caching[14]. These findings collectively indicate that NLP pipeline orchestration — particularly when multiple retrieval and processing stages are chained — is dominated by connection and coordination overhead rather than computation.

2.5 The Model Context Protocol

The Model Context Protocol (MCP), introduced by Anthropic in 2024, has rapidly emerged as the de facto standard for agent-tool integration. Ehtesham et al. survey four agent communication protocols — MCP, ACP, A2A, and ANP — and identify MCP's JSON-RPC client-server interface as the foundation for secure tool invocation and typed data exchange[15].

MCP defines two official transport mechanisms: stdio for local deployments and Streamable HTTP for remote deployments. The Streamable HTTP transport, introduced in specification version 2025-03-26, replaced the earlier HTTP+SSE transport from 2024 and operates through a single endpoint accepting HTTP POST requests with optional SSE streaming for server-initiated messages. By 2026, production deployments have surfaced scalability gaps: the MCP project has explicitly noted that 'stateful sessions fight with load balancers' and is evolving the transport and session model so that servers can scale horizontally without holding state[26]. WebSocket remains a supported custom transport within the MCP ecosystem — agent frameworks including Microsoft's Agent Framework provide MCPWebsocketTool as an integration point — but it is not the protocol's primary recommended transport for general-purpose remote deployments.

Venkiteela characterizes MCP as "the semantic complement to traditional API infrastructures," arguing that it enables "persistent context propagation, context-bound tool invocation, and embedded governance metadata across heterogeneous systems"[16]. Li and Xie provide critical analysis of MCP integration challenges, including "novel security vulnerabilities, privacy complexities, debugging difficulties across protocols, and the need for robust semantic

negotiation mechanisms"[17].

2.6 Context Window Memory Management

The treatment of the LLM context window as a memory system has received growing attention. Packer et al.[23] introduced MemGPT, which draws an explicit analogy between the LLM context window and a virtual memory system, proposing paging mechanisms to move information between a limited "main memory" (the context window) and external storage. While MemGPT demonstrated the viability of the operating-system metaphor, its implementation focused on conversational agents rather than tool-heavy enterprise workflows.

Recent empirical work on context-window memory management[21] extends this line of reasoning with a rigorous analysis of 857 production agentic sessions. The central finding is that the field treats the context window as the entire memory system, when in practice it functions as an L1 cache — small, expensive, and poorly managed. Three sources of structural waste are identified: unused tool schemas (11.0%), duplicated content (2.2%), and stale tool results (8.7%) that are reprocessed at a median amplification factor of 84.4x. The proposed four-level memory hierarchy — L1 (generation window), L2 (working set, demand-paged), L3 (session history, compressed), L4 (cross-session persistent memory) — provides a taxonomy within which the present work can be precisely situated. Specifically, the WebSocket-native NLP architecture described here operates at the L2/L3 boundary: it maintains a stateful working set of NLP artifacts (entity graphs, coreference chains, topic distributions) in connection-scoped RAM, thereby serving as an external backing store that the context-window manager can page into L1 on demand.

A key insight from this analysis is the "inverted cost model" of LLM memory: unlike traditional operating-system memory where keeping data resident is free and page faults are expensive, in the LLM regime keeping tokens in context incurs cost at every generation step while re-reading from an external store is a one-time expense. This inversion strengthens the case for tool-layer state management — if retaining NLP results in context is expensive, then maintaining them in an external WebSocket-scoped session and fetching only what is currently relevant represents the economically optimal strategy.

2.7 gRPC and Alternative Streaming Protocols

For server-to-server communication between agent processes and NLP microservices, gRPC with bidirectional streaming over HTTP/2 represents the primary technical alternative to WebSocket. gRPC provides Protocol Buffer schema enforcement with code generation, native HTTP/2 stream-level flow control via WINDOW_UPDATE frames, mature Kubernetes service mesh integration (Istio, Envoy), and strong language-agnostic tooling. Balu's comparative analysis finds that gRPC achieves 2.5x higher throughput than REST for streaming applications[9].

However, gRPC imposes constraints that limit its applicability to the use cases targeted by this paper. First, gRPC lacks native browser support without a proxy layer (grpc-web or Envoy), making it unsuitable for edge and embodied deployments where the agent runtime executes in a browser-based control interface. Second, gRPC does not support the subprotocol negotiation mechanism (RFC 6455, Section 1.9) that enables zero-downtime API versioning in WebSocket deployments (Section 4.4). Third, MCP's tool registration and discovery protocol operates over JSON-RPC, and while gRPC-to-MCP adapters exist, they introduce the same translation overhead that the WebSocket-native architecture eliminates. Finally, the connection-scoped state model

central to this architecture — where entity registries, coreference maps, and topic distributions persist implicitly in server memory — maps naturally to WebSocket's long-lived connection semantics but requires explicit state management in gRPC's call-scoped streaming model.

The choice between WebSocket and gRPC is deployment-context dependent. For high-throughput, Kubernetes-native microservice architectures with strict schema contracts and mature observability requirements, gRPC bidirectional streaming may be preferable. For browser-proximate edge deployments, MCP-native tool registration, and NLP pipelines requiring implicit cross-operation state propagation, WebSocket provides a more natural fit.

3. THE FIVE ARCHITECTURAL PAIN POINTS

We identify five systemic pain points that arise when enterprise agents invoke NLP operations through conventional REST/HTTP APIs.

3.1 Pain Point 1: Connection Overhead Accumulation

Each REST API call requires a full TCP+TLS handshake, which typically adds 100–300ms of latency before any application-level data transfer begins. This characterization applies to HTTP/1.1 deployments without connection pooling. Modern HTTP/2 clients with persistent connection pools amortize the TLS handshake across multiple requests, reducing per-request overhead to microsecond-range framing costs. However, even with HTTP/2 connection reuse, each NLP operation remains an independent request-response cycle: the server discards all processing state between calls, and interdependent operations cannot share intermediate results at the transport layer. For an agent executing N NLP operations per reasoning chain, the cumulative connection overhead is O(N) multiplied by the handshake cost. OpenAI's own documentation for their WebSocket mode states that "for rollouts with 20+ tool calls, we have seen up to roughly 40% faster end-to-end execution" when switching from REST to WebSocket, directly attributing this improvement to eliminated per-turn handshake overhead.

In a typical NLP analysis pipeline where an agent chains process_text, extract_entities, resolve_coreferences, extract_keywords, extract_topics, vectorize_text, and rerank_documents — 7 operations — the connection overhead alone contributes 700ms–2,100ms of pure waiting time. With a persistent WebSocket connection, this overhead is paid once (at connection establishment) and amortized across all subsequent operations.

3.2 Pain Point 2: Stateless Context Loss

REST APIs are stateless by design: each request must contain all information necessary for processing, and the server retains no session information between requests. For NLP operations, this creates a particularly severe inefficiency because operations are inherently interdependent. Entity extraction produces an entity graph that should inform coreference resolution; resolved coreference chains should shape topic modeling; and all preceding outputs should contextualize causal analysis.

In a stateless architecture, the agent must either: (a) re-send all preceding outputs with each subsequent request, increasing payload size and parsing overhead; or (b) maintain a server-side session store keyed by session identifiers, adding infrastructure complexity and introducing consistency challenges. Neither approach is architecturally natural. A persistent WebSocket

connection is inherently stateful — the server holds accumulated NLP context (entity maps, coreference clusters, topic distributions, expert personas) in connection-scoped memory, and each subsequent operation inherits this context automatically.

Empirical analysis of production context windows[21] quantifies the cost of this statelessness. Stale tool results — NLP outputs from prior reasoning steps that remain in the context window despite having no further relevance — are reprocessed at a median amplification factor of 84.4x: a tool result produced once is re-read by the model on average 84.4 subsequent turns before it is finally displaced. "Dead tool output" accounts for 26.5% of all measured context waste. A WebSocket-native NLP service that maintains its own session state eliminates this category of waste entirely. Instead of dumping raw NLP outputs into the context window where they accumulate and are reprocessed turn after turn, the agent queries the in-memory knowledge graph maintained by the NLP service and retrieves only what is currently relevant to the active reasoning step.

3.3 Pain Point 3: Polling-Induced Latency for Asynchronous Operations

Certain NLP operations — particularly model fine-tuning, deep causal analysis, and topic modeling over large corpora — are inherently long-running. In a REST architecture, the agent must either block (wasting reasoning cycles) or poll for completion (wasting API calls and introducing polling-interval latency). As noted in the enterprise agent failure analysis by the Composio team, the "Polling Tax" is one of three traps that kill agent pilots in production: "Polling doesn't scale. It wastes 95% of your API calls, burns through quotas, and never achieves real-time responsiveness."

WebSocket's full-duplex nature eliminates polling entirely. The server pushes progress events (training loss curves, intermediate analysis results, completion signals) directly to the agent over the open connection. The agent's execution loop can continue reasoning about other tasks and react to completion events when they arrive.

3.4 Pain Point 4: Absence of Backpressure

When an NLP service generates results faster than the agent's LLM can consume them — a common scenario during bulk document processing or when streaming enriched chunks from a semantic chunking operation — REST provides no mechanism for flow control. HTTP/2 partially addresses this through native stream-level flow control via WINDOW_UPDATE frames, providing per-stream backpressure semantics. However, this mechanism operates at the transport layer and does not extend to application-level flow control between logically related NLP operations multiplexed across independent HTTP/2 streams. The agent must accept the full response, buffer it in memory, and process it sequentially. For large outputs (e.g., a knowledge graph with 44 causal relations, 43 temporal relations, and 33 entities from a single document), this can exhaust agent-side memory.

WebSocket provides application-layer backpressure through the bufferedAmount property and drain events. When the agent's buffer fills, the server automatically throttles output. When the buffer drains, the server resumes. This prevents memory bloat on the agent side and keeps the pipeline stable under load — precisely the conditions that Ehtesham et al. identify as critical for enterprise deployment[15].

3.5 Pain Point 5: Protocol Alignment with Agent Standards

The Model Context Protocol supports WebSocket as a custom transport alongside its primary Streamable HTTP transport. While Streamable HTTP is MCP's recommended transport for general-purpose remote deployments, the stateful NLP pipeline use case — where cross-operation context must persist across multiple tool invocations within a session — aligns more naturally with WebSocket's persistent connection semantics. Agent frameworks including Microsoft's Agent Framework provide MCPWebsocketTool as a first-class integration point for exactly this class of stateful tool interaction.

When NLP services are served over stateless HTTP, an agent-side session management layer is required to track which NLP artifacts are current, which have been superseded, and which cross-operation dependencies exist — complexity that the WebSocket-native architecture handles implicitly at the transport layer. For deployment contexts where MCP's Streamable HTTP is preferred (e.g., horizontally scaled multi-tenant services), the architectural principles of connection-scoped state and cross-operation context propagation can be implemented through server-side session management, though this reintroduces some of the infrastructure complexity that the WebSocket-native approach eliminates.

4. THE PROPOSED ARCHITECTURE: WEBSOCKET-NATIVE NLP

4.1 Architectural Overview

We propose an architecture in which all NLP operations — sentence segmentation, entity extraction, coreference resolution, keyword analysis, topic extraction, vectorization, semantic chunking, cross-encoder reranking, expert persona generation, business intelligence orchestration, and deep causal analysis — are served through a single persistent WebSocket endpoint. The agent authenticates once at connection time (via Bearer token in the WebSocket handshake headers) and subsequently invokes any operation by sending a JSON message with an action field and a payload field.

The server maintains connection-scoped state: an entity registry, coreference cluster map, topic distribution cache, and expert persona store. Each operation reads from and writes to this shared state, enabling cross-operation context propagation without explicit state management by the agent.

4.2 Progressive Result Streaming

For orchestrated operations that invoke multiple sub-operations (e.g., a deep causal analysis that internally chains segmentation, entity extraction, coreference resolution, temporal analysis, and causal inference), the WebSocket connection enables progressive result streaming. Intermediate results are pushed to the agent as each sub-operation completes, allowing the agent to begin reasoning about partial outputs before the full pipeline finishes.

This directly implements the "tool partial execution" concept introduced by Xu et al.[7], but at the transport layer rather than the application layer. The NLP service does not need to expose partial execution APIs — the WebSocket connection naturally supports streaming of intermediate frames.

4.3 Multiplexed Parallel Operations

When an agent requests multiple independent NLP operations simultaneously (e.g., entity extraction, keyword analysis, and topic extraction on the same text), these operations are multiplexed over the single WebSocket connection. Each request carries a message_id, and responses are tagged with the corresponding identifier, allowing the agent to demultiplex results as they arrive.

In browser-based agent deployments and edge control interfaces, this avoids the per-domain connection limits (typically 6–7 HTTP connections) that constrain parallel REST calls. In server-side deployments, the benefit is primarily operational simplicity: a single multiplexed connection rather than a pool of parallel HTTP connections, and eliminates the per-connection resource overhead on the server side. For the fused parallel function calling pattern described by Singh et al.[5], WebSocket multiplexing provides a natural transport-level implementation.

4.4 Subprotocol Negotiation for API Versioning

The WebSocket protocol (RFC 6455, Section 1.9) supports subprotocol negotiation during the handshake. The client sends Sec-WebSocket-Protocol: blinkt-nlp-v2, blinkt-nlp-v1 and the server selects the highest mutually supported version. This enables zero-downtime API versioning without URL path changes, content negotiation headers, or breaking existing clients — a significant operational advantage for enterprise deployments where multiple agent versions may be in concurrent production.

4.5 Binary Frame Support for Efficient Embedding Transfer

WebSocket natively supports both text and binary frame types. For operations that produce high-dimensional numerical outputs — particularly vectorization (384-dimensional dense vectors) and reranking (relevance scores) — binary serialization via MessagePack or Protocol Buffers reduces payload size by 50–80% compared to JSON encoding and eliminates serialization/deserialization overhead on both sides.

4.6 Implications for Context Window Efficiency

The architectural choice to maintain NLP state at the tool layer has direct implications for the context window waste taxonomy identified in recent empirical work[21]. When NLP processing state persists in connection-scoped RAM on the WebSocket server, agents no longer need to carry entity graphs, coreference chains, or topic distributions inside the LLM's context window. These artifacts reside in the NLP service's session memory and are available on demand via lightweight query messages — a pattern that maps precisely to the L2 (working set) tier in the proposed memory hierarchy[21].

Consider the concrete case of an agent performing iterative document analysis. Under a stateless REST architecture, each reasoning step must inject the full entity graph and coreference map into the prompt so that the LLM can reason about previously identified entities. The measured cost of this pattern: dead tool output occupying context across turns is reprocessed at 84.4x median amplification. Under the WebSocket-native architecture, the entity graph lives in the NLP server's memory. The agent issues a targeted query — "retrieve entities related to the Acme acquisition" — and receives only the relevant subset, keeping context window consumption proportional to the active reasoning task rather than to the cumulative history of all prior NLP operations.

Within this memory hierarchy framework[21], the WebSocket-native NLP service functions as an external L2/L3 backing store. The context-window manager (L1) can page NLP artifacts in on demand and evict them when no longer needed, confident that no information is lost because the authoritative copy resides in the NLP service's session state. This decoupling reduces context-window pressure and enables the demand-paging strategies to operate over a substantially smaller residual working set.

4.7 Deployment in Embodied Agent Systems

Robotics and embodied AI represent a deployment domain where the consequences of stateless NLP infrastructure extend beyond latency and cost into the realm of physical safety. The AI robotics market is projected to grow from $6.11 billion to $33.39 billion by 2030 at a 40.4% compound annual growth rate[22], driven by the integration of large language models into robot control systems. As LLM-based agents assume responsibility for interpreting natural-language commands in physical environments, the statefulness of the underlying NLP infrastructure becomes a safety-critical requirement.

Consider a warehouse robot receiving the spoken command "cancel that." In a stateless NLP architecture, the coreference "that" cannot be resolved — the NLP service has no memory of the previous command and no entity graph tracking objects in the workspace. The robot must either fail silently (a safety hazard), request clarification (introducing latency in a time-sensitive environment), or hallucinate a referent (potentially dangerous). A WebSocket-native NLP service, by contrast, maintains a persistent entity graph that tracks objects, commands, and spatial relations across the entire interaction session. Coreference resolution operates over the full discourse history stored in connection-scoped memory, enabling accurate interpretation of anaphoric references without re-transmitting the entire command history.

The persistent session state provided by the WebSocket architecture is particularly well-suited to the demands of embodied AI: entity graphs tracking physical objects in the robot's workspace; coreference chains linking commands across multi-step manipulation tasks; and real-time state updates as the robot's environment changes through its own actions or external perturbations. These NLP artifacts must be maintained continuously and updated incrementally — a pattern that aligns naturally with the stateful, long-lived connections of the WebSocket protocol and is fundamentally incompatible with stateless REST semantics.

Integration with existing robotics infrastructure reinforces the protocol-level argument. ROS 2 (Robot Operating System 2), the dominant middleware for research and industrial robotics, already supports WebSocket bridges for web-based teleoperation and monitoring. NVIDIA's Isaac platform and Jetson edge-inference hardware target WebSocket-based communication for real-time sensor data and command streaming. Recent work on LLM-driven industrial robot control[24] has independently adopted WebSocket as the communication protocol between the language model and the robot controller, validating the architectural choice from an industrial-automation perspective. Emerging open-source frameworks such as Hugging Face LeRobot further demonstrate the convergence of language-model agents and robotic control systems — a convergence that demands stateful, low-latency NLP infrastructure at the tool layer.

5. ANALYSIS

5.1 Latency Model

Let H represent the TCP+TLS handshake overhead per connection (typically 100–300ms for HTTP/1.1 without connection reuse), N represent the number of NLP operations per reasoning chain, and C represent the number of reasoning chains per task.

Under HTTP/1.1 without connection pooling — the baseline that characterizes many current commercial NLP APIs — the total connection overhead per task is H × N × C. Under WebSocket, it is H (a single handshake at connection time). For a typical enterprise task with N=10 operations and C=5 chains, and H=150ms (a conservative estimate for TLS 1.3): REST/HTTP/1.1 connection overhead = 150 × 10 × 5 = 7,500ms (7.5 seconds). WebSocket connection overhead = 150ms.

Under HTTP/2 with persistent connection pooling, the per-request connection overhead drops to microsecond-range framing costs, substantially narrowing the raw connection-overhead advantage. In this baseline, WebSocket's primary latency benefits shift from eliminated handshakes to: (a) implicit state propagation between operations (eliminating payload re-transmission of prior NLP results), (b) progressive streaming of intermediate results within orchestrated pipelines, and (c) elimination of polling overhead for asynchronous operations.

The analytical estimate of 35–45% end-to-end latency reduction applies to the HTTP/1.1 baseline where connection overhead constitutes 40–50% of total tool-calling time — consistent with OpenAI's reported ~40% improvement for WebSocket mode in agentic workflows. Against an HTTP/2 baseline, the latency advantage is smaller but non-zero, driven primarily by state propagation and streaming benefits rather than connection overhead elimination. A controlled experiment quantifying the exact reduction against both baselines remains the critical validation step (Section 7).

5.2 State Propagation Model

In the REST model, propagating state from operation i to operation i+1 requires either: (a) the agent to include the output of operation i in the request for operation i+1 (increasing payload by $O(|state_i|)$ per call), or (b) a server-side session store with read/write latency and consistency guarantees. In the WebSocket model, state propagation is implicit: the server holds $state_i$ in connection-scoped memory and operation i+1 reads it directly. The communication cost of state propagation is zero, and the consistency model is trivially satisfied (single-connection, single-server).

5.3 Context Waste Reduction Estimate

The context-window waste taxonomy[21] enables a quantitative estimate of the waste reduction attributable to tool-layer state management. Of the 21.8% structural waste measured across 857 production sessions, stale tool results account for 8.7% and dead tool output accounts for 26.5% of the total waste budget. Under the WebSocket-native architecture, NLP tool results no longer persist in the context window between turns — they reside in connection-scoped server memory and are retrieved on demand. Conservatively assuming that NLP operations constitute 40–60% of an agent's tool calls (consistent with the 7–12 NLP operations per reasoning chain observed in enterprise deployments), the architecture targets 3.5–5.2 percentage points of the 8.7% stale-result waste and a proportional share of the dead-tool-output category. Combined with the 35–45% latency reduction from eliminated connection overhead, the architecture delivers improvements across both the time and token-cost dimensions of agent efficiency.

5.4 Compatibility with Pipeline Parallelism

The pipeline parallelism approach demonstrated by PipeRAG — overlapping retrieval and generation to achieve 2.6x speedup[12] — maps naturally to WebSocket's full-duplex capability. The agent can send the next operation request while still receiving streaming results from the current operation. In the REST model, the agent must wait for the full response before sending the next request, serializing the pipeline and eliminating parallelism opportunities.

6. IMPLICATIONS FOR ENTERPRISE AGENT DEPLOYMENT

6.1 Reduced Infrastructure Complexity

Enterprise agents currently require session management middleware, polling infrastructure, state serialization layers, and protocol translation adapters to bridge the gap between stateless NLP APIs and stateful agent loops. A WebSocket-native NLP architecture eliminates all four categories of middleware, reducing the infrastructure surface area and associated operational complexity.

6.2 MCP Transport Compatibility

MCP's current transport architecture centers on Streamable HTTP for remote deployments, with WebSocket available as a supported custom transport. For NLP services that require persistent cross-operation state — the central use case of this paper — WebSocket transport offers architectural advantages within the MCP framework: NLP operations register as MCP tools, and the persistent connection serves as both the MCP transport and the NLP service channel, achieving protocol unification for stateful tool interactions.

For deployment contexts where Streamable HTTP is preferred (e.g., horizontally scaled services behind load balancers), the cross-operation context propagation principles described in this paper can be implemented through server-side session management with explicit session identifiers, though this trades the implicit state advantage of WebSocket for the operational simplicity of stateless HTTP scaling.

6.3 Cost Efficiency

Usage-based billing models for NLP services (e.g., per-character or per-call pricing) are orthogonal to the transport protocol. However, the elimination of polling calls for asynchronous operations and the reduced payload sizes from binary frame encoding both contribute to lower effective costs per NLP operation. For enterprise agents processing large document corpora, the cumulative savings can be substantial.

6.4 Context Window Pressure Reduction

The measurement of 21.8% structural waste in production context windows[21] establishes a concrete cost baseline that WebSocket-native NLP directly addresses. Of the three waste categories identified — unused tool schemas (11.0%), duplicated content (2.2%), and stale tool results (8.7%) — the architecture presented here most directly targets the third. When NLP results persist in the tool layer's connection-scoped memory rather than being injected into the context window, the stale-result category is substantially reduced: the agent retrieves current, targeted results from the NLP service rather than carrying the full history of prior NLP outputs in context.

The reduction extends to the dead-tool-output subcategory, which accounts for 26.5% of all measured waste. In a stateless architecture, every NLP tool result — entity lists, topic distributions, coreference maps — enters the context window as a tool-response message and remains there for the duration of the session, consuming tokens at every subsequent generation step. The "inverted cost model"[21] makes this especially expensive: retaining a token in context costs money on every turn, while re-reading from an external store is a one-time expense. The WebSocket-native architecture exploits this inversion directly. NLP artifacts live in connection-scoped RAM — cheap to maintain, free to query — and enter the context window only when the agent's current reasoning step requires them. The system achieves a form of natural demand paging: the NLP service is the backing store, and the context window is the cache, populated only with actively referenced data.

7. LIMITATIONS AND FUTURE WORK

WebSocket connections require persistent server-side resources, which introduces scalability challenges compared to stateless REST endpoints. Load balancing persistent connections is more complex than distributing stateless requests. Connection migration (moving a session from one server to another during scaling events) requires explicit state serialization that partially negates the implicit state advantage.

The analysis in this paper is primarily architectural rather than empirical. While the latency model (Section 5.1) is grounded in well-characterized TCP+TLS handshake costs and consistent with OpenAI's reported measurements, the context-waste reduction estimates (Section 5.3) extrapolate from the underlying production data[21] under simplifying assumptions about the proportion of NLP-originated tool calls. A controlled experiment comparing end-to-end agent performance under REST and WebSocket NLP backends — measuring both wall-clock latency and token consumption per task — remains the critical validation step.

Future work should investigate: (1) the optimal granularity of connection-scoped state (per-document vs. per-task vs. per-session); (2) hybrid architectures where lightweight operations use REST and stateful pipelines use WebSocket; (3) formal benchmarking of end-to-end agent task completion time under WebSocket vs. REST NLP services; (4) the interaction between WebSocket-native NLP and speculative agent execution patterns such as those introduced by SPAgent[3]; (5) the integration of the tool-layer memory hierarchy proposed here with context-window demand-paging systems such as the demand-paging system described in [21], to empirically measure the combined waste reduction achievable when both layers cooperate; and (6) the deployment of WebSocket-native NLP services on edge hardware (e.g., NVIDIA Jetson) for latency-sensitive robotics applications where network round-trips to cloud-hosted REST endpoints are prohibitive.

7.1 Horizontal Scaling and State Management

Persistent WebSocket connections introduce first-order challenges for horizontal scaling. In Kubernetes deployments, sticky sessions (via session affinity or consistent hashing) are required to route reconnecting clients to the same server holding their connection-scoped state. During rolling deployments, connection-drain logic must gracefully migrate active sessions — a process that requires explicit state serialization to an external store (e.g., Redis), partially negating the implicit state advantage that motivates the architecture. KEDA-based autoscaling of WebSocket servers requires careful tuning of connection-per-pod thresholds and drain timeouts. For

enterprise environments with bursty agent workloads, the persistent resource cost of idle WebSocket connections may outweigh connection-overhead savings compared to stateless HTTP alternatives.

7.2 Security Considerations

The paper's security treatment is limited to Bearer token authentication during the WebSocket handshake (Section 4.1). Production deployments require additional mechanisms: token rotation during long-lived sessions without connection teardown necessitates a mid-session re-authentication protocol; mutual TLS (mTLS) for service-to-service authentication in zero-trust architectures; credential revocation procedures when a session is compromised; and per-tenant entity registry isolation in multi-tenant deployments to prevent cross-tenant data leakage through the shared connection-scoped state model. The persistent, accumulated entity state maintained by the NLP service also presents an expanded attack surface for prompt injection — adversarial inputs in earlier operations could poison the entity graph or coreference state consumed by subsequent operations within the same session.

7.3 Observability and Distributed Tracing

WebSocket connections do not integrate with OpenTelemetry's standard HTTP span instrumentation. Distributed tracing across multiplexed WebSocket messages requires custom span propagation — each message_id must carry trace context, and the NLP service must create child spans for each operation. This represents non-trivial instrumentation overhead compared to HTTP/2 REST APIs, where standard OpenTelemetry HTTP middleware provides tracing effectively without custom code. Organizations adopting the WebSocket-native architecture should budget for observability tooling investment.

8. CONCLUSION

The iterative, stateful, event-driven nature of LLM-based agent execution loops is fundamentally misaligned with the stateless, request-response semantics of REST/HTTP APIs. This paper has demonstrated that a persistent WebSocket connection serving a unified NLP microservices pipeline resolves five critical enterprise agent pain points: connection overhead accumulation, stateless context loss, polling-induced latency, backpressure absence, and protocol incompatibility with emerging standards. The architecture analytically estimates a 35–45% reduction in cumulative tool-calling latency, eliminates the need for session management middleware and polling infrastructure, enables progressive result streaming and pipeline parallelism, and aligns with the Model Context Protocol's supported WebSocket transport mode.

Beyond enterprise software, we have shown that the same architectural imperatives apply with even greater urgency in embodied AI systems, where stateless NLP infrastructure introduces not merely latency and cost penalties but physical safety hazards. The growing integration of LLM-based agents into robotic control systems[22][24] demands persistent, low-latency NLP infrastructure that maintains session state across the full duration of a physical task.

Taken together with concurrent work on context-window memory management[21], the architecture presented here describes one half of a complete memory hierarchy for agentic AI. The WebSocket-native NLP service provides stateful processing at the tool layer — connection-scoped RAM holding entity graphs, coreference chains, and topic distributions —

while context-window demand-paging systems manage eviction and retrieval of what remains in the LLM's working memory. As autonomous agents become the primary interface through which enterprises and embodied systems interact with NLP services, the transport architecture underlying those services becomes a first-order design decision. WebSocket-native NLP represents not merely a performance optimization but a fundamental alignment between service infrastructure and the computational model of its consumers.

REFERENCES

[1] He, Q., Nan, J., Jiao, J., et al. "Z-Space: A Multi-Agent Tool Orchestration Framework for Enterprise-Grade LLM Automation." arXiv:2511.19483, 2025. https://arxiv.org/abs/2511.19483

[2] Zeng, G., Chen, X., Hu, J., et al. "Routine: A Structural Planning Framework for LLM Agent System in Enterprise." arXiv:2507.14447, 2025. https://arxiv.org/abs/2507.14447

[3] Huang, Z., Zeng, W., Fu, T., et al. "Reducing Latency of LLM Search Agent via Speculation-based Algorithm-System Co-Design." arXiv:2511.20048, 2025. https://arxiv.org/abs/2511.20048

[4] Luo, M., Shi, X., Cai, C., et al. "Autellix: An Efficient Serving Engine for LLM Agents as General Programs." arXiv:2502.13965, 2025. https://arxiv.org/abs/2502.13965

[5] Singh, S., Karatzas, A., Fore, M., et al. "An LLM-Tool Compiler for Fused Parallel Function Calling." arXiv:2405.17438, 2024. https://arxiv.org/abs/2405.17438

[6] Hathidara, A., Yu, J., Schreiber, S. "Disambiguation-Centric Finetuning Makes Enterprise Tool-Calling LLMs More Realistic and Less Risky." arXiv:2507.03336, 2025. https://arxiv.org/abs/2507.03336

[7] Xu, Y., Kong, X., Chen, T., Zhuo, D. "Conveyor: Efficient Tool-aware LLM Serving with Tool Partial Execution." arXiv:2406.00059, 2024. https://arxiv.org/abs/2406.00059

[8] Badurowicz, M. and Lasocha, W. "Comparison of WebSocket and HTTP Protocol Performance." Journal of Computer Sciences Institute, 2021. https://doi.org/10.35784/jcsi.2452

[9] Balu, S. "gRPC Performance for Audio and Voice Streaming Applications: A Comparative Analysis with REST API and WebSockets." 2025. https://doi.org/10.46610/ijeitsec.2025.v001i02.005

[10] Kaminski, L., Kozlowski, M., Sporysz, D., et al. "Comparative Review of Selected Internet Communication Protocols." arXiv:2212.07475, 2022. https://arxiv.org/abs/2212.07475

[11] Hassan, M. "Choosing the Right Communication Protocol for Your Web Application." arXiv:2409.07360, 2024. https://arxiv.org/abs/2409.07360

[12] Jiang, W., Zhang, S., Han, B., et al. "PipeRAG: Fast Retrieval-Augmented Generation via Algorithm-System Co-design." arXiv:2403.05676, 2024. https://arxiv.org/abs/2403.05676

[13] Jin, C., Zhang, Z., Jiang, X., et al. "RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation." arXiv:2404.12457, 2024. https://arxiv.org/abs/2404.12457

[14] Lin, W. "Higress-RAG: A Holistic Optimization Framework for Enterprise Retrieval-Augmented Generation via Dual Hybrid Retrieval, Adaptive Routing, and CRAG." 2025.

[15] Ehtesham, A., Singh, A., Gupta, G.K., Kumar, S. "A Survey of Agent Interoperability Protocols: MCP, ACP, A2A, and ANP." arXiv:2505.02279, 2025. https://arxiv.org/abs/2505.02279

[16] Venkiteela, P. "The New Interoperability Paradigm: Model Context Protocol (MCP), APIs, and the Future of Agentic AI." Computer Fraud and Security, 2025. https://doi.org/10.52710/cfs.817

[17] Li, Q. and Xie, Y. "From Glue-Code to Protocols: A Critical Analysis of A2A and MCP Integration for Scalable Agent Systems." arXiv:2505.03864, 2025. https://arxiv.org/abs/2505.03864

[18] Moreno-Schneider, J., Bourgonje, P., Kintzel, F., Rehm, G. "A Workflow Manager for Complex NLP and Content Curation Pipelines." arXiv:2004.14130, 2020. https://arxiv.org/abs/2004.14130

[19] Jhandi, P., Kazi, O., Subramanian, S., Sendas, N. "Small Language Models for Efficient Agentic Tool Calling." arXiv:2512.15943, 2025. https://arxiv.org/abs/2512.15943

[20] Kandogan, E., Rahman, S., Bhutani, N., et al. "A Blueprint Architecture of Compound AI Systems for Enterprise." arXiv:2406.00584, 2024. https://arxiv.org/abs/2406.00584

[21] Mason, T. "The Missing Memory Hierarchy: Demand Paging for LLM Context Windows." arXiv:2603.09023v1, March 2026. https://arxiv.org/abs/2603.09023

[22] International Federation of Robotics. "Top 5 Global Robotics Trends 2026." January 2026. https://ifr.org/ifr-press-releases/news/top-5-global-robotics-trends-2026

[23] Packer, C., Wooders, S., Lin, K., et al. "MemGPT: Towards LLMs as Operating Systems." arXiv:2310.08560, 2023. https://arxiv.org/abs/2310.08560

[24] "LLM-driven agent for speech-enabled control of industrial robots." ScienceDirect, 2025. https://www.sciencedirect.com/science/article/pii/S2590123025027276

[25] Model Context Protocol Specification. "Transports — Streamable HTTP." Version 2025-11-25. https://modelcontextprotocol.io/specification/2025-11-25/basic/transports

[26] Model Context Protocol Blog. "Exploring the Future of MCP Transports." December 2025. https://blog.modelcontextprotocol.io/posts/2025-12-19-mcp-transport-future/