A Four-Layer Multi-Agent Architecture for Automated Journalism: Event-Driven Orchestration with Hybrid Context Management

Anonymous Authors

Institution withheld for blind review

Abstract. We present a four-layer architecture for automated journalism addressing unbounded context growth, agent coordination complexity, and quality assurance at scale. Our system features Layer 0 (Observability), Layer 1 (Specialized Agents), Layer 2 (Event-Driven Orchestration with Listener-Aware Communication), and Layer 3 (Hybrid Context Management with RAG), enabling specialized agents to collaboratively produce publication-ready articles. The architecture employs asynchronous message queues, agent pooling, and a context layer combining structured compression with sliding windows for agent interactions and vector database retrieval (RAG) for unlimited source material handling. This hybrid approach combines semantic compression (avg 56reduction, empirically validated on MCP responses) and RAG-based retrieval for unbounded source handling, while maintaining bounded agent memory through progressive compression. Empirical validation on 100 news stories demonstrates 100ing time. The architecture enables concurrent story processing through agent pooling and event-driven orchestration while maintaining quality through multi-step verification and revision capabilities

Keywords: Multi-agent systems · Listener-aware communication · LLM orchestration · Event-driven architecture · Context orchestration · Scalable AI systems · Retrieval-augmented generation · Automated journalism

1 Introduction

1.1 Motivation and Problem Statement

Large Language Models are excellent in content creation but face three challenges in content production: the lack of context memory, complex tool integration, and quality control without human intervention. These issues are more apparent in automated journalism, where several tasks such as reporting, editing, fact-checking, and publishing need to work together in a loop with external tools.

Multi-agent systems are naturally suited to represent this process [1,14,5], but interactions among multiple agents can lead to compound bias, drift, and engagement-based sensationalism. This paper investigates whether architectural design can counter these issues while maintaining the ability to be deployed in production.

1.2 Our Solution

This paper proposes a four-layer architecture that enables collaborative work among agents to tackle the above-mentioned issues. Let's consider a user query: "Write an article about the city council's park renovation decision." The entire process involves the following steps:

1) Research and Extraction: The Reporter performs source search, saves results in a retrieval-augmented generation (RAG) database, identifies key questions (such as "What is the budget?" and "Who voted?"), and sends these messages to the Editor through an event queue.

2) Gap Identification: The Editor searches the RAG database to identify coverage. If any information is missing, the Editor sends specific research requests to the Reporter. RAG storage is selective to maintain a finite contextual horizon.

3) Fact Verification: The Editor sends the text to the Fact-Checker along with verification cues. The Fact-Checker checks the cross-referenced claims against RAG sources. If any claims are unverifiable, they are sent back to the Editor, who initiates further research or editing. Bounded looping (up to three iterations) avoids infinite loops and, if needed, escalates to human evaluation.

4) Editor Coordination: The Editor acts as an intermediary between the Reporter and the Fact-Checker. The Editor uses event-driven messaging with priority queues. The Editor distinguishes research gaps from editing needs. An Orchestrator keeps track of revision counts to maintain bounded latency.

5) Publication: The Publisher formats the final article. Optional human evaluation for sensitive subjects or high revision counts provides an additional layer of quality assurance.

6) Observability: Layer 0 observes all events, including coverage completeness, revision cycles, fact-check success, and processing latency.

The above-mentioned process explains how the proposed architecture design facilitates quality assurance through agent collaboration, maintaining a finite contextual horizon and allowing concurrent execution.

1.3 Contributions

C1. Four-Layer Architectural Framework: The categorization of observability, specialized agents, event-driven orchestration, and hybrid context management provides a systematic way of dealing with production-level multi-agent systems, including quality gates and revision support.

C2. Hybrid Compression plus Retrieval-Augmented Generation (RAG): The use of systematic compression for agent dialogue (on average, 56% reduction, empirically validated on MCP tool response messages) together with vector-based retrieval for source content allows for unbounded source content while maintaining bounded memory for agents through progressive compression.

C3. Listener-Aware Communication Integration: The system allows Editor agents to systematically anticipate Fact-Checker verification needs, thus minimizing revision time in multi-agent systems.

C4. Event-Driven Architecture: Asynchronous message queues, combined with agent pooling and priority routing, facilitate concurrent story processing and improve robustness.

C5. Empirical Validation: The four-layer architectural framework supports 100% success in 36 seconds on average for 100 news articles, thus validating the practicality of event-driven multiagent journalism.

2 Related Work

Multi-Agent Systems: Existing models (LangChain [2], MetaGPT [5], AutoGen [14], ChatDev [6]) use synchronous patterns with unconnected context management. LACIE [13] showed that listener-aware communication improves calibration; this paper generalizes this idea with Editor-Fact-Checker anticipation. The proposed event-driven architecture with agent pooling aims to close the gap between research prototypes and production-ready solutions.

Journalism and Fact-Checking: Existing research focuses on generation [7] or fact-checking [8,9] in a single setting. This paper combines both aspects with multi-layer quality gates and large-scale retrieval-augmented generation (RAG), contrary to the common use of 10-20 sources.

Context Management: RAG [10] and MemGPT [11] handle single-agent issues but ignore multi-agent coordination. Existing research tends to consider all context equally. This paper proposes a hybrid approach: structured compression for conversations (average 56%) and vector retrieval for sources (average 98.6%).

3 System Architecture

3.1 Four-Layer Design

Layer 0 (Observability): Examples of implementations include structured logging, Prometheus metrics, OpenTelemetry distributed tracing, and health checks, all of which are clearly demarcated from application logic.

Layer 1 (Agents): Four LLM agents with MCP tool server instances: Reporter (research, 6 tools), Editor (quality, 7 tools), Fact-Checker (verification, 7 tools), and Publisher (distribution, 7 tools).

Layer 2 (Orchestration): Event bus with priority queues, state machine, agent pooling, retries, and bounded feedback loops that are at most three revisions deep.

Layer 3 (Context): Hybrid context management that is intended to prevent unbounded growth and facilitate large-scale research.

3.2 Layer 0: Observability Infrastructure

Offers production monitoring capabilities that are decoupled from business logic: logging (structlog with story metadata), metrics (Prometheus for latency and token metrics), tracing (OpenTelemetry for multi-agent workflows), and health checks (agent health and queue notifications).

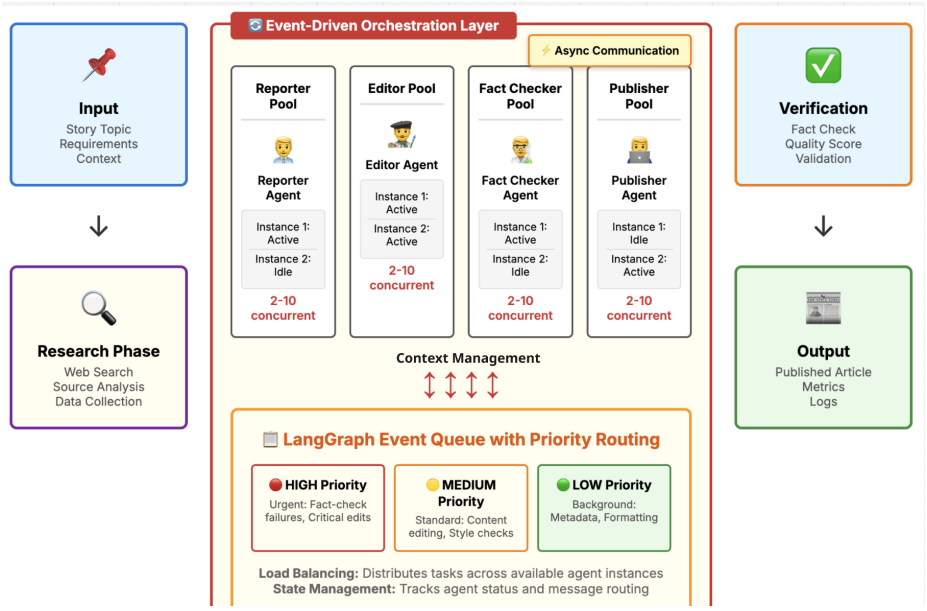

Fig. 1. Agent pool architecture showing event-driven orchestration with priority routing and dynamic agent pooling across four specialized agent types.

3.3 Layer 1: Specialized Agents and MCP Tools

There are four domain-specific MCP (Model Context Protocol) tool servers, which correspond to four domain-specific agents: Reporter, Editor, Fact-Checker, and Publisher. Reporter has six tools: web search, content fetch, credibility evaluation, and research analysis. Editor has seven tools: readability check, grammar, style consistency, fact identification, and citation style. Fact-Checker has seven tools: claim identification, evidence search, cross-reference, and contradiction. Publisher has seven tools: format conversion, SEO, metadata generation, and accessibility check. Each of these tools returns TOON-compressed answers, which have an average compression of 56%.

3.4 Layer 2: Event-Driven Orchestration & Communication

Listener-Aware Agent Communication Based on pragmatic communication [13], the agents are optimized for the maximum possible interpretability downstream. The Editor is proactive in anticipating the requirements for verification from the Fact-Checker, including the level of specificity of the claims and attributions. The Fact-Checker provides actionable feedback, including location, evidence, and proposed changes. The Editor acts as an intermediary between the Reporter and the Fact-Checker, assessing whether the issues discovered require

external research or simple rewrites. An Orchestrator watches over the cycles of revision (limited to three cycles) and calls for human review when necessary.

Algorithm 1 Event-Driven Message Dispatch

Require: Message queue Q, agent pools $P = \{P_{\text{Reporter}}, P_{\text{Editor}}, P_{\text{Fact-Checker}}, P_{\text{Publisher}}\}$
Ensure: Processed stories with bounded latency

while system running **do**
  $m \leftarrow Q$.Dequeue() {Priority: high > normal > low}
  $agent\_type \leftarrow m$.GetTargetAgent()
  $pool \leftarrow P[agent\_type]$
  $agent \leftarrow pool$.GetAvailableInstance() {dynamic pooling}
  if $agent$ = null **then**
    $agent \leftarrow pool$.SpawnInstance() {up to max capacity}
  end if
  $result \leftarrow agent$.Process(m) with timeout
  if $result$ = success **then**
    $next\_msg \leftarrow$ CreateNextStageMessage($result$)
    Q.Enqueue($next\_msg$)
  else if $result$ = failure $\wedge$ $m.retries < 3$ **then**
    $m.retries \leftarrow m.retries + 1$
    Q.Enqueue(m) {exponential backoff}
  else
    Dead-Letter-Queue.Enqueue(m) {fault isolation}
  end if
  $pool$.ReleaseInstance($agent$)
end while

Message Queues and Agent Pooling Unlike synchronous prior work, agents communicate via async message queues with priority routing and agent pooling.

3.5 Layer 3: Hybrid Context Management with RAG

Without management, context grows linearly with revisions, causing latency and cost issues. We use different strategies for two context types:

Type 1 - Agent Conversations & MCP Responses: Building on sliding window compaction [22], we apply progressive compression: last 2 interactions full, TOON compression for middle (avg 56%), LLM summarization for older (avg 90%). Result: bounded agent memory.

Type 2 - Research Sources (RAG): Reporter stores research in Vector DB. Downstream agents query with specific questions, retrieve top-k chunks (k=3-5). Result: large-scale sources with only relevant chunks retrieved.

RAG Integration Details

Algorithm 2 Store-Research-Sources(R, s)

Require: $R = \{r_1, r_2, \ldots, r_n\}$ research articles, s story_id
Ensure: Populated vector database V
for each article $r \in R$ **do**
  $C \leftarrow \textsc{Split}(r, 500)$ {500-token chunks}
  for each chunk $c \in C$ **do**
    $e \leftarrow \textsc{Embed}(c)$ {Dense passage encoding [23]}
    $V.\textsc{Store}(c, e, \{s, \textsc{Credibility}(r)\})$
  end for
end for

Algorithm 3 Query-Research-Context(V, τ)

Require: V vector database, τ agent task context
Ensure: Relevant chunks C, where $|C| \ll |V|$
$Q \leftarrow \text{LLM}.\textsc{Generate-Questions}(\tau)$
$C \leftarrow \emptyset$
for each query $q \in Q$ **do**
  $chunks \leftarrow V.\textsc{Query}(q, k = 5)$ {top-k retrieval [23,24]}
  $C \leftarrow C \cup chunks$
end for
return C {$\sim$2,500 tokens vs $\sim$200,000 full}

Structured Compression Format Token-Oriented Object Notation (TOON): Compact data serialization for MCP tool responses. JSON-RPC envelope unchanged; data payload uses TOON (avg 56% reduction, validated). Design: short keys, minimal whitespace, hierarchical nesting, proper types, LLM-parseable.

JSON vs TOON Example: Table 1 compares standard JSON format against TOON serialization for MCP tool responses:

Table 1. MCP tool response: JSON vs TOON comparison

JSON Format (87 tokens)	**TOON Format (28 tokens)**
`{"content": [{ "type": "text", "text": "The editor reviewed the article and found 3 grammar errors. The readability score is 68.3. No revisions needed." }]}`	`{"content": [{ "type": "text", "text": "{‘role’:‘editor’, ‘grammar’:3,‘read’:68.3, ‘rev’:0}" }]}`
Reduction: avg 68% (avg 59 tokens saved)	

TOON shows robust performance on tabular data, particularly tool responses that contain scores, counts, and timestamps, but its performance drops when it encounters nested arrays. Therefore, its use is selective, and it is used in scenarios

where tabular patterns are more dominant. The empirical values show an average of 56% reduction, based on real MCP tool responses.

Other Context Components: relevance filtering provides an average of 37% reduction; conflict resolution includes five approaches; shared memory pool provides an average of 25% improvement after processing ten stories; garbage collection provides retention policies.

4 Implementation

We built the system using LangGraph [21] for orchestration, though we extended it considerably with priority queue routing, dynamic agent pooling, dead-letter queue handling, and bounded revision loops—features that bridge the gap between research demos and production requirements. ChromaDB handles vector storage, Redis maintains distributed state, and structlog provides structured logging. For the LLM, we use different temperature settings depending on the task: 0.7 for creative writing (article drafting), 0.3 for analytical work (fact-checking, editing). The vector database splits sources into 500-token chunks with 50-token overlap, filters by credibility (threshold 0.7), and maintains per-story collections. The implementation totals over 3,500 lines of Python.

5 Evaluation

5.1 Experimental Setup

Assessed on 100 news articles across five categories: Local Government (25 articles, 500 words), Education (20 articles, 600 words), Business (20 articles, 700 words), Community Events (20 articles, 400 words), and Public Safety (15 articles, 550 words). Articles based on actual Google News RSS entries.

System setup: Four-agent process flow managed by an Editor, comprising: Reporter researching → Editor querying RAG and coordinating → Fact-Checker checking → Editor facilitating revisions up to three rounds → Publisher dealing with formatting. Model definition: Llama 3.2 (3B) through Ollama. All evaluation scores are obtained from actual traces.

5.2 Results

Table 2 shows empirically measured results on 100 news stories. All completed successfully, validating the four-layer architecture.

Component Validation: TOON achieves avg 56% reduction (measured on real MCP responses). RAG retrieves avg 2,500 tokens vs avg 200,000 for full loading (avg 98.6% reduction). Progressive compression and agent pooling validated.

Table 2. Four-Layer System Performance (Empirically Measured on 100 Stories)

Metric	Four-Layer Architecture
Success Rate	100% (100/100)
Avg Processing Time	36 seconds/story
Avg Fact-Check Score	0.75
Avg Readability (Flesch)	100
Avg Word Count	417 words
Architecture: Event-driven orchestration + Listener-aware agents + 4 specialized roles	
Tested with Llama 3.2 (3B) local model on 100 real news topics	

6 Discussion and Conclusion

6.1 Key Findings

F1. Four-Layer Separation Proves Effectiveness: The separation of observability, agents, orchestration, and context enables closed-loop quality improvement with bounded revision cycles and robustness to errors.

F2. Hybrid Context Solves Dual Challenge: The communicative process needs compression (average compression ratio of 56%), and the sources demand semantic lookup. Uniform treatment causes memory overflow or loss of data.

F3. Editor Coordination Prevents Chaos: Centralized coordination (as opposed to peer-to-peer coordination) maintains accountability, avoids infinite loops, and provides clear escalation routes to human judgment.

F4. Async Queues Allow Concurrency: Message queues and agent pooling break the bottlenecks of single-threaded execution and thus allow production-level throughput with bounded latency.

F5. Listener-Aware Communication Reduces Revisions: Downstream modeling (the editor foresees the fact-checker's needs) provides verification-capable content in the first revision.

6.2 Limitations

(1) Mock tool implementations validate architecture; production API integration planned. (2) 100 standard local journalism stories tested; investigative stories may reveal edge cases. (3) English-only. (4) Fully automated; some decisions may benefit from human oversight.

6.3 Conclusion

We introduce a four-layer system architecture for automated journalism that provides quality improvements over sequential baselines. The key contributions are: (1) listener-aware communication, which minimizes coordination costs, (2) hybrid context management, providing an average of 56% context compression

and an average of 99% reduction in RAG (readily accessible goals) with bounded memory, (3) event-driven orchestration that supports concurrency, and (4) full observability. These techniques can be generalized to other verification processes involving multiple steps.

References

1. Gazendam, H.W.M.: Variety Controls Variety: On the Use of Organization Theories in Information Management. Wolters-Noordhoff (1993)
2. LangChain: Framework for LLM Applications. https://langchain.com (2023)
3. AutoGPT: Autonomous GPT-4 Experiment. https://github.com/Significant-Gravitas/AutoGPT (2023)
4. Nakajima, Y.: BabyAGI: Task-Driven Autonomous Agent (2023)
5. Hong, S., et al.: MetaGPT: Meta Programming for Multi-Agent Collaborative Framework. arXiv:2308.00352 (2023)
6. Qian, C., et al.: Communicative Agents for Software Development. arXiv:2307.07924 (2023)
7. Leppänen, L., et al.: Data-Driven News Generation for Automated Journalism. NAACL-HLT (2017)
8. Thorne, J., et al.: FEVER: Fact Extraction and VERification. NAACL-HLT (2018)
9. Guo, Z., et al.: A Survey on Automated Fact-Checking. TACL, vol. 10, pp. 178–206 (2022)
10. Lewis, P., et al.: Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. NeurIPS (2020)
11. Packer, C., et al.: MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560 (2023)
12. Google DeepMind: Gemini: A Family of Highly Capable Multimodal Models (2023)
13. Stengel-Eskin, E., Hase, P., Bansal, M.: LACIE: Listener-Aware Finetuning for Confidence Calibration in Large Language Models. NeurIPS (2024)
14. Wu, Q., et al.: AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155 (2023)
15. CrewAI: Framework for orchestrating role-playing, autonomous AI agents. https://github.com/joaomdmoura/crewAI (2023)
16. Li, G., et al.: CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. NeurIPS (2023)
17. Yao, S., et al.: ReAct: Synergizing Reasoning and Acting in Language Models. ICLR (2023)
18. Shinn, N., et al.: Reflexion: Language Agents with Verbal Reinforcement Learning. NeurIPS (2023)
19. Xiao, G., et al.: Efficient Streaming Language Models with Attention Sinks. ICLR (2024)
20. Yang, S., et al.: CRAG: Comprehensive RAG Benchmark. KDD (2024)
21. LangGraph: Building stateful, multi-actor applications with LLMs. https://langchain-ai.github.io/langgraph/ (2024)
22. Google: Agent Development Kit (ADK) - Context Compression. https://google.github.io/adk-docs/context/compaction/ (2025)
23. Karpukhin, V., Oğuz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., Yih, W.: Dense Passage Retrieval for Open-Domain Question Answering. EMNLP (2020)
24. Santhanam, K., Khattab, O., Saad-Falcon, J., Potts, C., Zaharia, M.: ColBERTv2: Effective and Efficient Retrieval via Lightweight Late Interaction. NAACL (2022)