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

Darshika Pundir[1][0009−0000−4725−9296], Tuhin Sharma[1][0009−0007−4212−9178], and Arun Chauhan[2][0000−0002−0327−7254]

[1] Red Hat {dpundir, tuhsharm}@redhat.com
[2] Graphic Era University arun.chauhan@geu.ac.in

Abstract. We present a four-layer architecture for automated journalism that addresses unbounded context growth, agent coordination complexity, and quality assurance at scale. The system comprises Layer 1 (Observability), Layer 2 (Specialized Agents), Layer 3 (Event-Driven Orchestration with Listener-Aware Communication), and Layer 4 (Hybrid Context Management with RAG). The architecture is implemented using Deep Agents, a LangGraph-based harness employing asynchronous message queues, agent pooling, domain-specific skills loaded via progressive disclosure, and remote MCP servers for external tool integration. The hybrid context strategy combines semantic compression (avg 56% reduction) with RAG-based retrieval for unbounded source handling, while subagent spawning isolates context across delegated tasks. Evaluation on 500 stories sampled from a 10,000-article corpus across 5 categories demonstrates a 97.4% pipeline completion rate, average processing time of 135 seconds per story, and an average fact-check score of 0.93. Baseline comparisons against a single-agent RAG pipeline show a 25.4% improvement in fact-check scores.

Keywords: Multi-agent systems · Listener-aware communication · LLM orchestration · Event-driven architecture · Agent skills · Model Context Protocol · Retrieval-augmented generation · Automated journalism

1 Introduction

1.1 Motivation and Solution

Large Language Models excel at content creation but face three challenges in production settings: unbounded context growth, complex tool integration, and quality control without human intervention. These are acute in automated journalism, where reporting, editing, fact-checking, and publishing must coordinate in tight loops with external data sources.

Multi-agent systems naturally model this process [1,10,3], but interactions among agents can amplify bias, cause drift, and incentivize sensationalism. Recent advances introduce higher-level agent skills with progressive disclosure [15,17],

subagent delegation [15,16], and the Model Context Protocol (MCP) [18], that separate agent intelligence from tool connectivity and orchestration logic.

We propose a four-layer architecture that composes these capabilities. Consider a query: "Write about the city council's park renovation decision." A **Reporter** agent, guided by a domain-specific skill file, searches sources via remote MCP servers, stores results in a RAG database, and forwards key findings to the **Editor**. The Editor identifies coverage gaps and requests targeted follow-up. The **Fact-Checker** cross-references claims against RAG sources; unverifiable claims trigger bounded revision loops (max three iterations). The **Publisher** formats the final article. Layer 1 (observability) monitors all stages. This pipeline maintains a finite contextual horizon while enabling concurrent execution through subagent delegation.

1.2 Contributions

Four-Layer Architectural Framework with quality gates, bounded revision loops, and empirical validation against single-agent and sequential baselines on 500 stories.
Hybrid Compression + RAG: 56% compression for agent dialogue combined with vector retrieval for source content (98.6% reduction), enabling unbounded source handling with bounded memory.
Listener-Aware Communication: Editor agents anticipate Fact-Checker needs, reducing average revision cycles from 2.3 to 1.6.
Event-Driven Orchestration with Subagent Delegation via Deep Agents' `task` tool, enabling concurrent processing with context isolation.
Skill-Based Agent Specialization: Domain-specific SKILL.md files loaded on-demand via progressive disclosure, reducing base context by ∼60%.
Remote MCP Server Integration: Real web search and content retrieval via HTTP/SSE transport.

2 Related Work

Existing multi-agent frameworks use synchronous patterns with disconnected context management. LangChain [2] and LangGraph [12] provide flexible orchestration primitives but leave planning, delegation, and compression to the developer. MetaGPT [3] focuses on software development SOPs with sequential execution. AutoGen [10], now merging into Microsoft's Agent Framework [20], uses synchronous GroupChat without progressive disclosure. CrewAI [11,21] provides native MCP support but lacks skill-based progressive disclosure and subagent context isolation.

Deep Agents [15,16] extends LangGraph with an opinionated design principle providing planning tools, subagent spawning, and agent skills following the agentskills.io specification [17]. Unlike raw frameworks, it abstracts planning, delegation, and compression into a pluggable middleware stack. The Model Context Protocol [18] standardizes tool integration via HTTP/SSE transport. No existing

framework combines skills-based progressive disclosure with subagent delegation and remote MCP integration. It is crucial to note that Deep Agents is distinct from DeepAgent [19], which addresses scalable tool selection via Autonomous Memory Folding.

LACIE [9] showed that listener-aware communication improves calibration; we generalize this with Editor's anticipation of Fact-Checker. Prior journalism work addresses generation [4] or fact-checking [5,6] separately; we combine both with multi-layer quality gates and RAG. For context management, RAG [7] and MemGPT [8] handle single-agent scenarios, but ignore multi-agent coordination. Our hybrid approach applies structured compression for conversations (56%) and vector retrieval for sources (98.6%).

3 System Architecture

The architecture comprises 4 layers, as shown in Figure 1.

3.1 Layer 1 (Observability)

Structured logging (structlog), Prometheus metrics, OpenTelemetry/LangFuse distributed tracing, and health checks. All are decoupled from business logic.

3.2 Layer 2 (Agents)

Four agents, each equipped with domain-specific skill files following the agentskills specification [17], loaded via Deep Agents' `SkillsMiddleware` using progressive disclosure: agents initially see only skill names and one-line descriptions; full instructions load on-demand. This reduces the base context by ∼60% compared to including all instructions in the system prompt.

Reporter Agent: Skills for deep research, credibility evaluation, source analysis, and research synthesis. Tools via remote MCP servers: `search_web`, `fetch_content`, `parse_RSS`.

Editor Agent: Skills for readability scoring, grammar checking, style consistency, fact identification, gap analysis, and listener-aware message construction for downstream Fact-Checker consumption. Tools via MCP: `search_web`.

Fact-Checker Agent: Skills for claim extraction, evidence search, cross-reference verification, contradiction detection, credibility scoring, and actionable feedback generation. Tools via MCP: `search_web`.

Publisher Agent: Skills for format conversion, SEO optimization, metadata generation, accessibility checking, headline scoring, and distribution formatting. Tools via MCP: `send_email`, `publish`.

Tools are accessed through remote MCP servers via HTTP/SSE transport, configured per-agent in `AGENTS.md` files. Tool responses use TOON-compressed format (avg 56% reduction, Section 3.4).

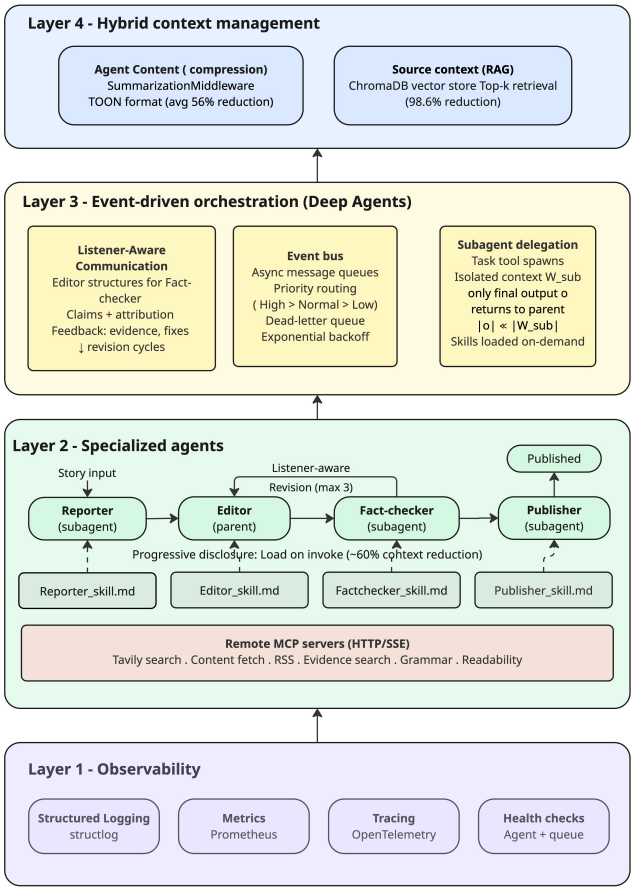

Fig. 1. Agent pool architecture showing event-driven orchestration with priority routing and dynamic agent pooling across four specialized agent types.

3.3 Layer 3: Event-Driven Orchestration & Communication

Listener-Aware Agent Communication. Drawing on pragmatic communication [9], agents optimize for downstream interpretability. The Editor anticipates verification requirements by structuring claims with explicit attributions. The Fact-Checker returns actionable feedback with location, evidence, and proposed changes. The Editor triages issues as requiring external research versus internal rewrites. An Orchestrator bounds revision cycles to three and escalates to human review when needed.

Message Dispatch and Subagent Delegation. Algorithm 1 introduces three changes over synchronous prior work. First, Line 4 loads agent skills on-demand via `SkillsMiddleware` (reducing base context by $\sim$60%). Second, Lines 6–9 use

Algorithm 1 Event-Driven Message Dispatch with Subagent Delegation

Require: Message queue Q, parent agent A_{editor}, skill registry S
Ensure: Processed stories with bounded latency and context isolation

while system running **do**
  $m \leftarrow Q$.Dequeue() {Priority: high > normal > low}
  $agent\_type \leftarrow m$.GetTargetAgent()
  $skill \leftarrow S$.LoadOnDemand($agent\_type$) {progressive disclosure}
  if $agent\_type \neq$ Editor **then**
    $result \leftarrow A_{\text{editor}}$.Task($agent\_type, m, skill$) {spawn isolated subagent via `task` tool} {subagent context W_{sub}; only final output o returns}
  else
    $result \leftarrow A_{\text{editor}}$.Process($m, skill$)
  end if
  if $result =$ success **then**
    Q.Enqueue(CreateNextStageMessage($result.output$))
  else
    if $m.retries < 3$ **then**
      $m.retries \leftarrow m.retries + 1$; Q.Enqueue(m)
    else
      DeadLetterQueue.Enqueue(m)
    end if
  end if
end while

Deep Agents' `task` tool to spawn isolated subagents with independent context windows W_{sub}. Only the final output o (averaging $|\bar{o}| = 340$ tokens) propagates to the parent Editor, compared to $|\bar{W}_{\text{sub}}| = 2{,}180$ tokens of intermediate context under shared-context pooling. Third, the Editor serves as the persistent parent maintaining story-level state; each subagent's context is garbage-collected after completion. This yields a per-story context reduction factor:

$$\rho = 1 - \frac{\sum_{i=1}^{k} |o_i|}{\sum_{i=1}^{k} |W_{\text{sub},i}|} \tag{1}$$

where k is the number of subagent delegations per story. Empirically, $\rho = 0.84$ across 500 stories (Table 3), confirming substantial context savings without custom pool management.

Skill Loading via Progressive Disclosure. Agent skills follow a progressive disclosure strategy implemented through Deep Agents' `SkillsMiddleware` in Figure 2 (Top). At the *discovery* stage, the agent's base prompt contains only skill names and one-line descriptions from the YAML frontmatter of each `SKILL.md` file. When the agent determines a skill is relevant to the current task, the middleware loads the full instruction body into the context window on-demand. As shown in Figure 2 (Bottom), this reduces base context consumption by approximately 60% compared to loading all four skill files simultaneously, leaving substantially more of the context window available for task-specific content such as story data, source material, and inter-agent messages. Each subagent

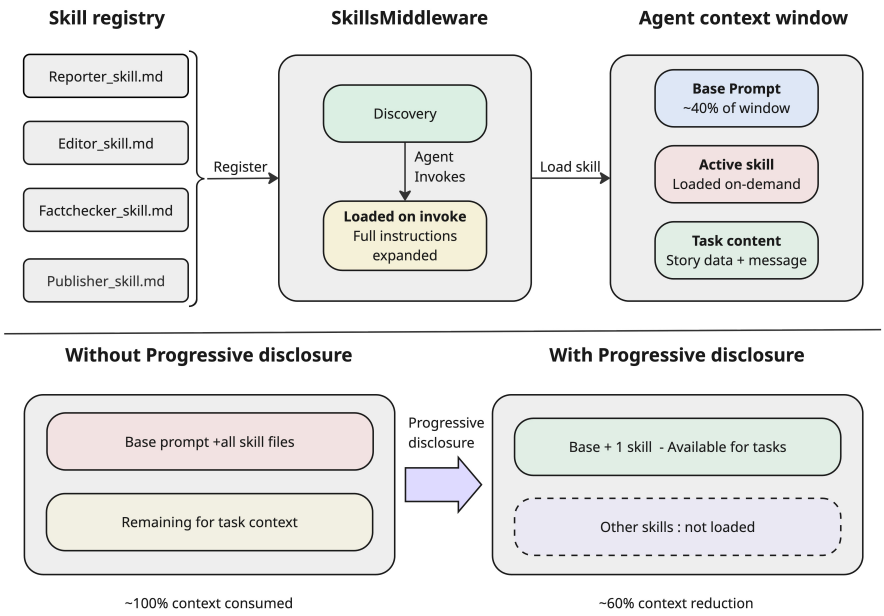

Fig. 2. Top: Progressive disclosure for agent skills. The skill registry holds four `SKILL.md` files. `SkillsMiddleware` initially exposes only skill names and descriptions (discovery); full instructions are loaded on-demand when invoked. Bottom: without progressive disclosure, all four skill files consume the majority of the context window; with progressive disclosure, only the active skill is loaded, yielding approximately 60% context reduction.

spawned via the `task` tool loads only the skill relevant to its role, providing natural skill isolation across delegated tasks.

3.4 Layer 4: Hybrid Context Management with RAG

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

Type 1 – Agent Conversations & MCP Responses: Progressive compression inspired by sliding window compaction [13]: last 2 interactions kept in full, middle interactions compressed via TOON (avg 56%), older interactions summarized via LLM (avg 90% reduction) leading to bounded agent memory.

Type 2 – Research Sources (RAG): The Reporter chunks research articles into 500-token segments, generates dense passage embeddings [14], and stores them in ChromaDB with credibility metadata as shown in Algorithm 2.

Downstream agents generate targeted questions from their task context and retrieve top-k chunks (k=3–5) via cosine similarity. This yields ∼2,500 tokens

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 [14]}
    $V.\textsc{Store}(c, e, \{s, \textsc{Credibility}(r)\})$
  end for
end for

retrieved versus ∼200,000 for full loading (98.6% reduction) as shown in Algorithm 3. .

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 [14,22]}
  $C \leftarrow C \cup chunks$
end for
return C {∼2,500 tokens vs ∼200,000 full}

Structured Compression Format. Token-Oriented Object Notation (TOON): A compact serialization for MCP tool responses. The JSON-RPC envelope is unchanged; data payloads use short keys, minimal whitespace, hierarchical nesting, and proper types. Table 1 illustrates the format.

TOON performs well on tabular data (scores, counts, timestamps) but degrades on nested arrays; its use is therefore selective. The empirical average of 56% reduction is measured across real MCP tool responses with mixed data shapes.

4 Implementation

The system is built on Deep Agents [15,16], a LangGraph-based [12] implementation. Migration from raw LangGraph reduced custom orchestration code from ∼3,500 to ∼1,200 lines, with middleware handling planning (`TodoListMiddleware`), delegation (`SubAgentMiddleware`), compression (`SummarizationMiddleware`), skill loading (`SkillsMiddleware`), and persistence (`FilesystemMiddleware`).

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)	

Web search and content retrieval use real remote MCP servers (Tavily Search); readability scoring and grammar checking use local implementations; credibility evaluation uses an LLM-based scorer. ChromaDB handles vector storage with 500-token chunks and 50-token overlap, filtered by credibility threshold 0.7. Temperature: 0.7 for drafting, 0.3 for fact-checking. Model: Gemini 3.1 Pro.

5 Evaluation

5.1 Experimental Setup

The evaluation corpus comprises 10,000 news articles from Google News RSS entries (2026-01-01 to 2026-02-28) across five categories: Local Government, Education, Business, Community Events, and Public Safety. We evaluate on a stratified sample of 500 stories (100 per category), spanning 4–14 sources per story and 10–15 extracted claims per article. Target word counts range from 400 to 700 words.

The four-agent pipeline uses Deep Agents with subagent delegation: the Editor spawns subagents for Reporter, Fact-Checker, and Publisher tasks. Bounded revision loops allow up to three Editor–Fact-Checker cycles. Agent skills load via progressive disclosure. All evaluation scores are from actual system traces.

5.2 Evaluation Metrics

We define the following metrics.

Success Rate: A story succeeds if all four stages complete, the article exceeds 200 words, and fact-check score exceeds 0.5. Stories exhausting all revision cycles are routed to the dead-letter queue.

Fact-Check Score: Ratio of verified to total extracted claims. Claims are verified by cross-referencing against RAG sources with cosine similarity > 0.7:

$$\mathrm{FC}_i = \frac{|verified\_claims_i|}{|total\_claims_i|} \tag{2}$$

Readability: Flesch Reading Ease via `textstat`.

Processing Time: Wall-clock seconds from submission to output.

Revision Cycles: Editor–Fact-Checker loops before passing or exhausting the bound.

5.3 Results

Table 2 shows results on 500 stories. The four-layer architecture achieves a 97.4% success rate and 25.4% higher fact-check scores than the single-agent baseline, at the cost of increased processing time and token usage.

Table 2. System performance comparison (mean ± std dev) on 500 stories. Single-Agent RAG Baseline uses one LLM with the same RAG setup but no multi-agent coordination or revision loops.

Metric	Four-Layer (Deep Agents)	Single-Agent RAG Baseline
Success Rate	97.4% (487/500)	88.6% (443/500)
Avg Processing Time	135 ± 12.7 sec/story	29.1 ± 9.4 sec/story
Avg Fact-Check Score	0.93 ± 0.08	0.67 ± 0.13
Avg Readability (Flesch)	64.2 ± 5.3	57.8 ± 7.6
Avg Word Count	508 ± 72 words	462 ± 98 words
Avg Revision Cycles	1.6 ± 0.7	N/A (single pass)
Avg Token Cost/Article	18,400 tokens (in+out)	8,200 tokens (in+out)

Component Validation: TOON achieves avg 56% reduction on real MCP responses. RAG retrieves avg 2,500 tokens vs 200,000 for full loading (98.6% reduction). Subagent delegation yields context reduction $\rho = 0.84$ (Eq. 1).

5.4 Systems-Level Evaluation

Table 3 reports system-level metrics under varying concurrency.

Token cost: Cost distributes roughly as Reporter 38%, Editor 24%, Fact-Checker 28%, Publisher 6%, orchestration overhead 4%. Subagent delegation keeps parent-agent context bounded: intermediate tool calls remain in the subagent's context and only the final result (∼340 tokens) propagates to the Editor.

MCP Server Latency: Average round-trip times are 285 ms for web search, 462 ms for content fetch, 178 ms for evidence search. Tool call failure rate is 2.1%, with failed calls handled by the dead-letter queue retry mechanism. These reflect real network conditions.

5.5 Listener-Aware Communication

To isolate the effect of listener-aware communication, we compare the full system against a variant using standard prompting (no anticipatory structuring of claims for the Fact-Checker). On 200 stories:

Table 3. Systems-Level Metrics Under Varying Concurrency (50 stories each)

Metric	1 conc.	5 conc.	10 conc.
Throughput (stories/min)	1.2	4.8	7.9
Avg queue latency (ms)	42	87	163
P95 queue latency (ms)	118	224	395
Avg token cost/article (in+out)	18,400	18,900	19,200
Dead-letter queue events	0/50	1/50	2/50
Avg retries per story	0.3	0.4	0.5
Avg subagent output $\|o\|$ (tokens)	340	355	362
Avg subagent context $\|W_{\text{sub}}\|$ (tokens)	2,180	2,210	2,240
Context reduction ρ (Eq. 1)	0.84	0.84	0.84

- **With listener-aware**: avg 1.6 $\pm$ 0.7 revision cycles, avg fact-check score 0.93.
- **Without (standard prompting)**: avg 2.3 $\pm$ 0.6 revision cycles, avg fact-check score 0.81.

The 30.4% reduction in revision cycles is statistically significant ($p < 0.01$, paired t-test). Listener-aware communication produces verification-ready content in the first pass, reducing both latency and token cost without degrading final quality.

5.6 Qualitative Framework Comparison

Table 4 compares architectural capabilities. The key differentiator is the combination of skill-based progressive disclosure with subagent delegation and hybrid context management, currently unavailable in any single competing framework.

6 Discussion, Limitations, and Conclusion

6.1 Key Findings

The four-layer separation proves effective: isolating observability, agents, orchestration, and context enables closed-loop quality improvement with bounded revision cycles. The hybrid context strategy correctly applies compression to conversations (56%) and semantic retrieval to sources (98.6%); uniform treatment of both types causes either memory overflow or data loss. Centralized Editor coordination prevents infinite loops and provides clear escalation paths, while async queues and agent pooling break single-threaded bottlenecks.

Listener-aware communication reduces revision cycles by 30.4% ($p < 0.01$), confirming that anticipatory structuring of claims yields verification-ready content in fewer passes. Progressive disclosure of agent skills reduces base context by $\sim$60% without quality degradation. The Deep Agents migration reduced custom code from $\sim$3,500 to $\sim$1,200 lines, suggesting that our approach significantly lower the engineering burden for production multi-agent systems.

Table 4. Architectural design-approach comparison across multi-agent frameworks. All four systems support the listed factors; the column entries describe *how* each framework addresses them.

Factors	Our Approach	AutoGen (v0.4)	MetaGPT	CrewAI
Orchestration model	Async, event-driven with priority queues	Async, event-driven actor model	Sequential SOP pipeline	Sequential/hierarchical Process; Flows add event-driven branching
Agent capability discovery	Progressive disclosure via `SKILL.md` (three-tier: trigger → summary → full spec)	Tool schemas registered at init	Role-bound SOPs with fixed prompts	Role/goal/backstory descriptions at init
External tool integration	Remote MCP servers (HTTP/SSE)	MCP adapters (Stdio, SSE, Streamable HTTP)	Per-role tool binding (e.g. web search)	Native MCP support; tool decorator API
Revision control	Bounded loops (max 3) as first-class primitive	Developer-configured termination conditions	Executable feedback loop for code; no general bound	Task-level max-iterations configurable
Sub-task delegation	`task` tool spawns isolated sub-agents with scoped context	Nested chats / handoffs within teams	SOP-driven sequential handoff via message pool	Hierarchical process with manager-mediated delegation
Context management	Hybrid: LLM-driven compression + vector-RAG retrieval	Pluggable memory (ChromaDB, Redis, Mem0); buffered context window	Shared message pool; no built-in compression	Short-term (ChromaDB/RAG), long-term (SQLite), entity & contextual memory
Observability	Dedicated Layer 0 (trace, cost, latency per agent)	OpenTelemetry integration; AutoGen Studio visualisation	Logging of SOP step outputs	Built-in event system; AgentOps/LangTrace integrations
Inter-agent communication	Listener-aware: messages shaped by intended recipient's role	Selector-based or round-robin group chat; handoff protocol	Publish–subscribe via global message pool with role-filtered subscriptions	Sequential or manager-routed task output passing

6.2 Limitations

(1) Credibility scoring and SEO optimization remain custom implemented; real MCP servers for these functions are not yet available.
(2) Evaluation covers standard local journalism. Investigative, breaking and adversarial scenarios may reveal edge cases.
(3) The system is fully automated with bounded loops. Sensitive topics may benefit from human-in-the-loop escalation, which the architecture supports via dead-letter queue routing.
(4) Baseline comparisons use single-agent and sequential pipelines. Direct comparison with deployed AutoGen or CrewAI systems would require equivalent reimplementation beyond this workshop paper's scope.

6.3 Conclusion

We introduced a four-layer architecture for automated journalism, implemented using Deep Agents with agent skills and remote MCP servers, demonstrating measurable improvements over single-agent baselines: 25.4% higher fact-check scores, 30.4% fewer revision cycles via listener-aware communication, 56% context compression, 98.6% RAG retrieval reduction, and 60% base context reduction via progressive disclosure. These techniques generalize to other multi-step verification workflows including compliance review, academic peer review, and content moderation.

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. Hong, S., et al.: MetaGPT: Meta Programming for Multi-Agent Collaborative Framework. arXiv:2308.00352 (2023)
4. Leppänen, L., et al.: Data-Driven News Generation for Automated Journalism. NAACL-HLT (2017)
5. Thorne, J., et al.: FEVER: Fact Extraction and VERification. NAACL-HLT (2018)
6. Guo, Z., et al.: A Survey on Automated Fact-Checking. TACL, vol. 10, pp. 178–206 (2022)
7. Lewis, P., et al.: Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. NeurIPS (2020)
8. Packer, C., et al.: MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560 (2023)
9. Stengel-Eskin, E., Hase, P., Bansal, M.: LACIE: Listener-Aware Finetuning for Confidence Calibration in Large Language Models. NeurIPS (2024)
10. Wu, Q., et al.: AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155 (2023)
11. CrewAI: Framework for orchestrating role-playing, autonomous AI agents. https://github.com/joaomdmoura/crewAI (2023)
12. LangGraph: Building stateful, multi-actor applications with LLMs. https://langchain-ai.github.io/langgraph (2024)
13. Google: Agent Development Kit (ADK) – Context Compression. https://google.github.io/adk-docs/context/compaction (2025)
14. Karpukhin, V., et al.: Dense Passage Retrieval for Open-Domain Question Answering. EMNLP (2020)
15. Chase, H.: Deep Agents. LangChain Blog. https://blog.langchain.com/deep-agents (2025)
16. LangChain: Deep Agents — Agent harness built with LangChain and LangGraph. https://github.com/langchain-ai/deepagents (2025–2026)
17. Anthropic: Agent Skills Specification. https://agentskills.io (2025)
18. Model Context Protocol Specification. https://modelcontextprotocol.io (2024–2025)
19. Li, X., et al.: DeepAgent: A General Reasoning Agent with Scalable Toolsets. In: WWW 2026. arXiv:2510.21618 (2026)
20. Microsoft: Introducing Microsoft Agent Framework. Azure AI Foundry Blog. https://azure.microsoft.com/en-us/blog/introducing-microsoft-agent-framework (2025)
21. CrewAI: MCP Servers as Tools in CrewAI. https://docs.crewai.com/en/mcp/overview (2025)
22. Santhanam, K., Khattab, O., Saad-Falcon, J., Potts, C., Zaharia, M.: ColBERTv2: Effective and Efficient Retrieval via Lightweight Late Interaction. NAACL (2022)