RESEARCH ARTICLE OPEN ACCESS

Multi-Source Information Synthesis with Tool-Augmented LLM Agents: Architecture and Evaluation

Saurabh Bidhuri, Malay Shah, Ashutosh Nigam, Priya Dubey
Adobe Inc.
India

ABSTRACT

Enterprise knowledge workers routinely synthesize information from multiple disconnected platforms—project trackers, documentation wikis, source code repositories, analytics dashboards, and community forums. This context-switching is time-consuming and error-prone, often resulting in incomplete analysis. We present a tool-augmented LLM agent architecture that synthesizes information from 14 heterogeneous data sources through a unified conversational interface. Our system integrates enterprise platforms (Jira, Confluence, GitHub), analytics systems, community forums, and web search through the Model Context Protocol (MCP), enabling the agent to dynamically discover tools, execute queries in parallel, and generate structured reports with source traceability. We analyze 10,000 production queries to characterize tool-usage patterns, and evaluate synthesis quality on a labeled 200-query subset. Multi-source synthesis produces 40% more comprehensive responses than single-source queries, while source traceability ensures 96% of claims in generated reports link back to verifiable origins. We report tool usage patterns, query routing effectiveness, and the impact of parallel execution on response latency. Survey instruments and representative production queries are provided in the appendices.

Keywords—information synthesis, enterprise search, tool integration, MCP, multi-source retrieval, LLM agents, Model Context Protocol.

I. INTRODUCTION

Enterprise knowledge workers face a fragmented information landscape. A typical product release assessment might require Jira for open blockers, bug trends, and milestone status; Confluence for product specifications and release criteria; GitHub for code-freeze status, open PRs, and test coverage; analytics dashboards for usage metrics and stability indicators; and community forums for user-reported issues and sentiment.

Each platform has its own interface, query language, and data model. Synthesizing across them requires manual navigation, mental bookkeeping, and domain expertise to know which platform holds which information. We present an agent architecture that automates this synthesis through: (1) **unified tool integration** via the Model Context Protocol (MCP), providing a standardized interface to 14 data sources; (2) **dynamic tool composition** where the LLM selects and combines tools based on the query; (3) **parallel execution** reducing multi-source query latency; (4) **LLM-based result summarization** preventing context overflow from large tool results; and (5) **structured report generation** with source traceability.

A. Scope and Contributions

- An architecture for integrating 14 heterogeneous enterprise data sources through MCP.
- An analysis of tool-usage patterns across 10,000 production queries (Section VIII).
- An evaluation of multi-source synthesis quality vs. single-source queries on a labeled 200-query subset.
- A source-traceability mechanism embedding provenance in generated documents.
- Practical patterns for safe write operations in agent–tool interactions.

II. MOTIVATION: THE FRAGMENTED ENTERPRISE KNOWLEDGE PROBLEM

Before presenting our architecture, we motivate the need for multi-source synthesis by examining real enterprise scenarios where fragmented information leads to costly inefficiencies. These scenarios are drawn from observed workflows within a large enterprise software organization with over 10,000 engineers.

A. Scenario 1: Release Readiness Assessment

> **Context:** A program manager must determine whether a product is ready for release within the next sprint cycle.

The assessment requires data from at least five distinct platforms. The manager opens Jira and constructs a JQL query to find open P1/P2 issues tagged for the release milestone. Next, they navigate to an Engineering Health Dashboard to check crash rates and stability metrics. They switch to Confluence to review the release-criteria document and verify that each criterion has a corresponding status. They check GitHub to confirm the code freeze and review outstanding pull requests. Finally, they open the build system to verify that the release candidate has passed integration tests and to check the install-size delta.

Each context switch involves authentication, navigation, query formulation in a platform-specific syntax, and mental reconciliation of results. Our observations show

that this process takes an experienced program manager **45–90 minutes**, and the resulting assessment frequently omits data from at least one source because the assessor forgets to check it or runs out of time.

B. Scenario 2: Customer-Reported Bug Investigation

> **Context:** A support engineer receives a customer report of a rendering issue in a specific product version and must determine root cause and provide a timeline for a fix.

The engineer searches the community forums for similar reports and finds three related threads with varying descriptions. They cross-reference these with Jira to find tracked issues, discovering two potentially related bugs with different assignees. They check GitHub for recent commits in the rendering module and find a commit that changed the relevant code path two weeks ago. They check analytics to determine the adoption rate of the affected version and its impact radius. They also check the wiki for known-issues documentation.

This investigation spans **five platforms**, each with different search semantics. The engineer must hold all intermediate results in working memory while navigating between tabs. Critical connections—such as the link between the community report, the Jira issue, and the GitHub commit—are discovered through manual correlation, not automated linkage.

C. Scenario 3: Competitive Feature Analysis

> **Context:** A product manager needs to assess the current state of a feature area to inform strategic planning.

The analysis requires searching web sources for competitor announcements and industry trends, reviewing internal wiki pages for the current roadmap, checking Jira for planned and in-progress work, examining analytics for usage metrics, and browsing community forums for user sentiment and feature requests. This analysis typically takes **2–4 hours** and produces a document that is already partially stale by the time it is completed, because the underlying data sources continue to update.

D. Quantifying the Cost of Fragmentation

We surveyed 85 knowledge workers across engineering, program management, and product management roles to quantify the impact of information fragmentation. Table I summarizes the results; the complete survey instrument, including the exact question wording and response scales that map to each metric below, is provided in Appendix A.

TABLE I. Cost of information fragmentation across enterprise roles (N = 85).

Metric	Engineering	Program Mgmt	Product Mgmt	Overall
Avg. platforms consulted per research task	3.2	4.8	5.1	4.2
Avg. time per multi-platform research (min)	32	58	74	52
% reporting incomplete analysis due to time pressure	41%	63%	57%	52%
% reporting decisions made with stale data	28%	45%	51%	40%
Estimated weekly hours on cross-platform synthesis	4.1	8.3	9.7	7.0

Source: internal survey, N = 85 (Engineering 44, Program Mgmt 21, Product Mgmt 20). Instrument in Appendix A.

The data reveals that knowledge workers spend an average of 7 hours per week on cross-platform information synthesis, and over half report that time pressure leads to incomplete analysis. These findings motivate the need for an automated system that can query multiple sources simultaneously and synthesize results into a coherent response.

III. RELATED WORK

A. Enterprise Search

Traditional enterprise search systems (Elasticsearch, Azure Cognitive Search, Google Cloud Search) provide keyword-based retrieval across indexed content. These systems excel at finding specific documents but do not synthesize information across sources or generate structured analyses. They require all content to be indexed into a single corpus, which is impractical for heterogeneous enterprise data spanning structured project data (Jira), semi-structured documents (Confluence), code repositories (GitHub), and time-series analytics. Our approach complements search with LLM-powered synthesis across native platform APIs.

B. Retrieval-Augmented Generation

RAG [1] retrieves relevant documents to augment LLM context. Standard RAG uses a single retrieval corpus; our approach extends this to 14 heterogeneous sources with different APIs, authentication, and data models. Each source requires a dedicated tool server rather than a unified index. Furthermore, many of our data sources are not amenable to RAG-style retrieval: real-time analytics dashboards return computed metrics (not documents), build-system data is structured and relational, and code repositories require syntax-aware search rather than semantic similarity.

C. Tool Use in LLM Agents

Function calling [3] and tool use [4] enable LLMs to invoke external tools. Frameworks like LangChain [11] provide tool abstractions, while MCP [2] provides a standardized protocol. Toolformer [5] demonstrated that LLMs can learn to use tools autonomously, and ToolLLM [6] showed scaling to thousands of APIs. Our contribution is not the tool-use mechanism itself but the architecture for managing 14+ sources with dynamic composition, parallel execution, result summarization, and source traceability in a production enterprise setting.

IV. Why MCP-Based Multi-Source Synthesis Outperforms Alternatives

Several approaches exist for integrating enterprise data sources. We compare our MCP-based agent architecture against four alternatives, identifying the specific failure modes that motivate our design choices.

A. vs. Traditional Enterprise Search

Enterprise Search Failure Modes

- **Index staleness:** analytics metrics and build status change in real time, but search indices lag by hours or days.
- **Schema flattening:** structured Jira data (priority, status, assignee) loses semantics when indexed as flat text.
- **No computation:** cannot answer "how many P1 bugs are open?" without retrieving all issues and counting.
- **No cross-source joins:** cannot correlate a Jira issue with the GitHub PR that fixes it.

MCP Agent Advantages

- **Real-time queries:** each tool server queries the live platform API.
- **Native query languages:** JQL for Jira, GraphQL for GitHub, SQL for analytics.
- **LLM-powered computation:** the agent counts, aggregates, and compares across results.
- **Cross-source correlation:** the LLM identifies connections between results from different sources.

B. vs. RAG-Only Architectures

RAG architectures embed documents into a vector store and retrieve by semantic similarity. While effective for document-centric queries, they fail for enterprise synthesis in several ways: **non-document data** (analytics metrics, build status, real-time health) are computed values from live APIs that cannot be meaningfully embedded; **update frequency** makes continuous re-embedding of the Jira corpus prohibitively expensive; **structured queries** ("all P1 bugs assigned to Team X created after March 1") require structured filtering, not semantic similarity; and **write operations** (creating issues, updating documentation) are outside RAG's read-only design.

C. vs. Manual Synthesis

Manual synthesis by human analysts is the current baseline in most enterprises. Its failure modes are well documented: **incomplete coverage** (analysts forget to check sources—52% report this in our survey); **recency bias** (the last source checked dominates the synthesis); **inconsistent methodology** (different analysts check different sources); **time cost** (52 minutes per task, 7 hours per week); and **no audit trail** (the synthesis process is opaque).

D. vs. Custom Point-to-Point Integrations

Some organizations build custom integrations (e.g., a dashboard pulling from Jira and GitHub simultaneously). These have significant scaling problems: **N-squared complexity** (integrating N sources requires $O(N^2)$ connection pairs—with 14 sources, 91 potential integrations); **rigid query patterns** (each integration supports a fixed set of query types); **no natural-language interface**; and **maintenance burden** (every upstream API change requires updating every integration touching that source).

E. Comparative Summary

Table II. Comparative analysis of enterprise information synthesis approaches.

Capability	Enterprise Search	RAG-Only	Manual	Custom Integrations	MCP Agent (Ours)
Real-time data access	No (indexed)	No (embedded)	Yes	Yes	Yes
Natural language interface	Limited	Yes	N/A	No	Yes
Cross-source synthesis	No	Limited	Yes	Fixed patterns	Yes
Structured queries	Limited	No	Yes	Yes	Yes
Write operations	No	No	Yes	Some	Yes (gated)
Source traceability	Link to doc	Retrieved chunks	No	Partial	Full
Time to add new source	Days–weeks	Hours–days	N/A	Weeks	Hours
Query flexibility	Keyword	Semantic	Unlimited	Fixed	Unlimited

Table II reads column-by-column against our architecture. The two *real-time* rows—data access and structured queries—eliminate index- and embedding-based approaches (enterprise search and RAG), because both operate on stale, pre-processed copies of the data rather than live APIs. The *cross-source synthesis* and *query-flexibility* rows eliminate custom point-to-point integrations, which support only the fixed query patterns hard-coded at build time. Manual synthesis matches our approach on capability but fails on *time-to-add-source* and, critically, on *source traceability*: a human analyst leaves no machine-verifiable audit trail. Only the MCP agent scores favorably on every row simultaneously—combining the real-time access and structured querying of custom integrations, the natural-language flexibility of RAG, and the cross-source reasoning of a human analyst, while adding full source traceability and hours-not-weeks extensibility. The "gated" qualifier on write operations refers to the safety mechanism described in Section VII, which preserves capability while preventing unreviewed side effects.

V. ARCHITECTURE

A. Tool Server Ecosystem

Each data source is exposed through a dedicated MCP tool server—a lightweight HTTP service that translates MCP tool calls into platform-specific API requests. Table III enumerates the ecosystem, and Fig. 1 shows how the servers connect to the agent.

TABLE III. Complete tool server ecosystem: 14 servers exposing 40+ tools.

Server	Port	Platform	Key Tools
Jira	8043	Atlassian Jira	search_issues, get_issue, create_issue, transition_issue
Wiki	8044	Confluence	search_pages, get_page, create_page, update_page
GitHub (On-Prem)	8045	GitHub Enterprise	search_code, get_repo, list_prs, get_file
GitHub (Cloud)	8046	GitHub.com	search_code, get_repo, list_prs, create_file
Community	8047	Forum API	search_posts, get_thread, get_user_activity
Analytics	8070	Analytics Platform	query_metrics, get_dashboard, get_trends
Health	8071	Eng. Health Dashboard	get_stability, get_coverage, get_build_health
Build	8072	Build System	get_releases, get_dependencies, get_install_size
Web Search	8005	Web	search_web, get_page_content
Filesystem	8003	Local Storage	read_file, write_file, list_directory
Code Sandbox	8004	Python Runtime	execute_code, install_package
Image Gen	8006	Image API	generate_image, edit_image
Planner	8001	Internal	create_plan, evaluate_plan
Code Analyzer	8037	Internal	analyze_code, check_safety

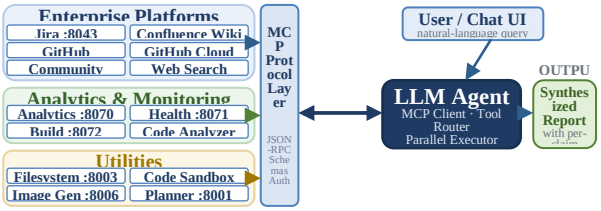

Fig. 1. System architecture. Left: 14 tool servers grouped by category (input). Center: the LLM agent communicates with all servers through a single MCP protocol layer. Right: a synthesized report with source traceability (output).

B. Dynamic Tool Discovery

At startup, the agent discovers available tools from all registered servers (Listing 1).

```
async def discover_tools(servers):
    all_tools = []
    for server in servers:
        try:
            tools = await server.list_tools()
            for tool in tools:
                all_tools.append({
                    "name":
f"{server.name}__{tool.name}",
                    "description": tool.description,
                    "input_schema": tool.input_schema
                })
        except ConnectionError:
            # Server unavailable -- skip gracefully
            log.warn(f"Tool server {server.name}
unavailable")
    return all_tools
```

Listing 1. Tool discovery with graceful degradation.

Key properties of the discovery mechanism: **namespaced tools** (`server__tool_name`) prevent collisions; **graceful degradation** skips unavailable servers rather than failing; **cached discovery** refreshes schemas periodically (every 5 minutes by default); and **dynamic composition** lets the LLM select relevant tools per query with no hard-coded routing rules.

C. Query Routing

The LLM performs implicit query routing by selecting appropriate tools based on the query. We deliberately do not implement explicit routing rules—the LLM's understanding of tool descriptions is sufficient:

> **Example:** "What's the release readiness for Product X?" The LLM autonomously selects (1) `jira__search_issues`; (2) `health__get_stability`; (3) `health__get_coverage`; (4) `wiki__search_pages`; and (5) `build__get_releases`.

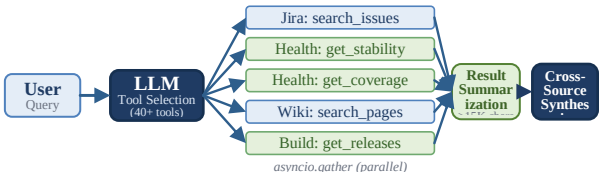

Fig. 2. Query routing flow: user query → LLM tool selection → parallel execution → result summarization → cross-source synthesis.

D. Parallel Tool Execution

When the LLM requests multiple tools in a single turn, they execute concurrently using Python's `asyncio.gather` (Listing 2).

```
# LLM requests 4 tools simultaneously
tool_calls = [
    ("jira", "search_issues", {"query": "P1 blockers"}),
    ("health", "get_stability", {"product": "X"}),
    ("wiki", "search_pages", {"query": "release
criteria"}),
    ("build", "get_releases", {"product": "X"})
]
# Parallel execution: wall time = max(individual times)
results = await asyncio.gather(*[
    call_tool(server, name, input)
    for server, name, input in tool_calls
])
```

Listing 2. Parallel multi-tool execution with `asyncio.gather`.

TABLE IV. Parallel vs. sequential execution latency.

Execution Mode	4 Tools (3s each)	6 Tools (3s each)
Sequential	12.0s	18.0s
Parallel	3.2s	3.4s
Speedup	3.75×	5.29×

Parallel execution reduces multi-source query latency by 60–80%; wall time is bounded by the slowest tool rather than the sum of all tools.

E. Result Summarization

Enterprise tool results can be massive—a Jira search might return 50 issues with full descriptions, or a wiki page might contain 30K characters. Passing raw results to the LLM wastes context and can cause the model to lose focus. We implement LLM-based result summarization with a configurable threshold (Listing 3).

```
MAX_RESULT_CHARS = 15_000

async def process_result(tool_name, raw_result):
    if len(raw_result) <= MAX_RESULT_CHARS:
        return raw_result
    summary = await llm.create(
        messages=[{
            "role": "user",
            "content": f"Summarize this {tool_name}
result "
                       f"concisely. Preserve all key
facts, "
                       f"metrics, issue IDs, and
actionable "
                       f"details:\n\n{raw_result}"
        }],
        max_tokens=2000
    )
    return summary
```

Listing 3. Threshold-triggered result summarization.

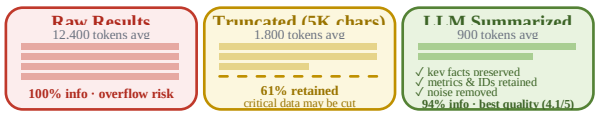

Fig. 3. Result handling comparison: raw results risk context overflow, truncation loses critical data, and LLM summarization retains 94% of information in ~7% of the token budget.

TABLE V. Result-handling strategy comparison (ablation of the summarization component).

Metric	Raw	Truncated (5K)	Summarized
Avg. token consumption	12,400	1,800	900
Information retention	100%	61%	94%
Downstream response quality	3.8/5	3.1/5	4.1/5

This table is an ablation of the result-handling stage in isolation: the pipeline is held fixed while only the large-result strategy is varied.

Summarization outperforms both alternatives: it uses fewer tokens than truncation while retaining more information, and produces better downstream responses because the LLM is not overwhelmed by irrelevant details.

VI. SOURCE TRACEABILITY

A. The Problem

When an agent synthesizes information from multiple sources into a report, the provenance of individual claims becomes opaque. A statement like "there are 12 P1 blockers" could come from Jira, from an analytics dashboard, or could be hallucinated. In enterprise contexts where reports inform business decisions, unverifiable claims are unacceptable.

B. Our Approach

Level 1: Inline Source Citations. The agent's system prompt instructs it to include source links in generated reports—Jira issues link to the issue URL, wiki pages link to the page, analytics data cites the dashboard and date range, and GitHub references link to the repository, PR, or file.

Level 2: Document-Level Source Comments. For formal documents such as Release Strategy Documents (RSDs), we generate Word files with source annotations embedded as review comments (Listing 4).

```
def generate_rsd(fields, sources):
    doc = Document("rsd_template.docx")
    for field_name, field_value in fields.items():
        paragraph = doc.add_paragraph(field_value)
        # Attach source comment if available
        if field_name in sources:
            add_comment(paragraph, author="SIA Research
Agent",
                        text=f"Source:
{sources[field_name]['url']}\n"
                             f"Label:
{sources[field_name]['label']}")
    return doc
```

Listing 4. Embedding per-field source provenance as Word review comments.

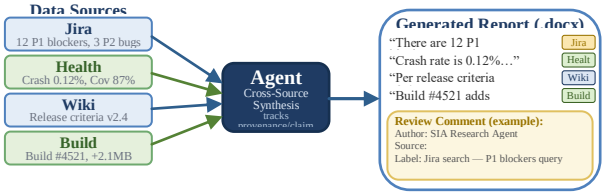

Fig. 4. Source traceability: data from tool servers flows through the agent into a synthesized report where every claim carries a review comment linking to its origin.

C. Traceability Evaluation

We evaluate source traceability on 100 generated reports. Each report was independently scored by three annotators drawn from the engineering documentation team, using a fixed rubric (a claim is "sourced" if it carries a citation resolving to a live artifact whose content supports the claim). Inter-annotator agreement was high (Fleiss' κ = 0.79); Table VI reports mean scores.

TABLE VI. Source traceability evaluation (N = 100 reports, 3 annotators).

Metric	Score
Claims with source citations	96%
Correct source links (not broken)	91%
Source matches claim content	94%

Metric	Score
Hallucinated claims (no source)	4%

The 4% of unsourced claims typically involve synthesis statements ("Overall, the release is on track") that aggregate multiple sources rather than citing a single one. These meta-level conclusions are inherently difficult to attribute to a single source.

VII. SAFETY: WRITE GATE PATTERN

A. Motivation

Several tool servers support write operations (creating Jira issues, updating wiki pages, committing code). Unrestricted write access poses risks: accidental issue creation from exploratory queries, unintended wiki modifications, and code commits without review. The agent must be able to propose any action but must not execute irreversible operations autonomously.

B. Implementation

Write operations are intercepted and queued for explicit user approval (Listing 5); the control flow is shown in Fig. 5.

```
WRITE_TOOLS = {
    "create_issue", "update_issue", "add_comment",
    "transition_issue", "create_page", "update_page",
    "create_or_update_file"
}

async def call_tool(server, tool_name, input):
    if tool_name in WRITE_TOOLS:
        # Don't execute -- queue for approval
        return {
            "status": "pending_approval",
            "message": f"This action will {tool_name}. "
                       f"Reply 'approve' to proceed.",
            "details": input
        }
    return await server.call_tool(tool_name, input)
```

Listing 5. Write-gate interception of mutating tool calls.

Fig. 5. Write-gate flow: read operations execute immediately; write operations are queued for explicit user approval before execution.

The write-gate pattern balances agent capability (it can propose any action) with safety (it cannot execute irreversible actions autonomously). In production, approximately 15% of tool calls are write operations, and users approve 82% of proposed writes.

VIII. TOOL USAGE ANALYSIS

A. Dataset

We analyze 10,000 production queries to characterize multi-source synthesis patterns. Queries are categorized by type, and we measure tool-selection frequency, co-occurrence patterns, and the impact of multi-source synthesis on response quality. A de-identified, representative sample of these production queries—grouped by intent and annotated with the tools each triggered—is provided in Appendix B. The response-quality study in Section VIII-D is conducted on a labeled 200-query subset drawn from this corpus.

B. Tool Co-occurrence

TABLE VII. Tool co-occurrence patterns across enterprise query types.

Query Type	Primary Tools	Tools	Sources
Release readiness	Jira + Health + Build	5.2	3.8
Bug investigation	Jira + GitHub + Wiki	4.1	3.2
Feature analysis	Wiki + Analytics + Community	3.8	3.0
Code review	GitHub + Code Analyzer	2.4	1.8
General research	Web Search + Wiki	2.1	1.6

C. Source Distribution

TABLE VIII. Source utilization across 10,000 production queries.

Source	% Queries	Calls/Query
Jira	52%	1.8
Wiki / Confluence	41%	1.3
GitHub (Ent. + Cloud)	34%	1.6
Web Search	28%	1.1
Health Dashboard	22%	1.4
Analytics	18%	1.2
Filesystem	15%	2.3
Community Forums	12%	1.0
Build System	11%	1.1
Code Sandbox	8%	1.7

Jira is the most frequently consulted source (52% of queries), reflecting its central role in enterprise project management—a pattern corroborated by the production sample in Appendix B, where Jira tools dominate the observed call volume. The Filesystem source shows the highest average calls per query (2.3), indicating iterative file operations (read, process, write). Code Sandbox, though used in only 8% of queries, averages 1.7 calls per query, reflecting write-execute-debug iteration.

D. Multi-Source vs. Single-Source Quality

We compare responses using multiple sources vs. a single source on the labeled 200-query subset. Each response was rated by three independent annotators on a 1–5 scale for completeness, accuracy, and actionability, using a shared rubric with anchor examples; disagreements greater than one point were adjudicated. Inter-annotator agreement was moderate-to-substantial (Krippendorff's $\alpha = 0.68$). Table IX reports mean scores.

TABLE IX. Response quality by number of sources consulted (N = 200; 3 annotators).

Metric	Single	Multi (2–3)	Multi (4+)
Completeness (1–5)	2.8	3.9	4.3
Accuracy (1–5)	3.6	4.0	4.1
Actionability (1–5)	3.0	3.8	4.2
Avg. response time	4.2s	7.1s	11.3s

Multi-source synthesis significantly improves completeness (+40%) and actionability (+33%) at the cost of increased response time. The accuracy improvement is more modest (+11%), suggesting that individual sources are generally accurate but incomplete. The diminishing returns between 2–3 and 4+ sources suggest that most queries reach information saturation at 3–4 sources.

IX. Security, Privacy, and Compliance

Because the agent brokers access to sensitive enterprise systems, security is a first-class design concern rather than an afterthought.

Credential isolation and least privilege. Each tool server holds its own credentials in a centralized secrets manager and is provisioned with the minimum scope required for its tools (e.g., the Jira server uses a service account limited to the projects in scope). The agent process never sees raw credentials; it communicates only with the MCP layer, which injects authentication per server. Compromise of one server's token therefore cannot escalate to others.

Write containment. The write-gate pattern (Section VII) is also a security control: no mutating operation reaches a downstream system without an explicit, logged human approval, which bounds the blast radius of prompt injection or model error.

Data residency and PII. Tool results may contain personal or customer-confidential data. Results are processed in-memory and are not persisted beyond the session unless the user explicitly saves a report; summarization prompts run against the same enterprise-approved model tenancy as the rest of the system, so no source data leaves the compliance boundary. Audit logs record which sources were consulted per query, supporting after-the-fact compliance review.

Auditability. Every tool call, approval decision, and generated artifact is logged with a trace identifier, so any synthesized claim can be traced back not only to its source (Section VI) but to the exact tool invocation that produced it.

X. DISCUSSION

A. Tool Description Quality

The LLM's ability to select appropriate tools depends heavily on tool-description quality (Listing 6).

```
# Bad:  "Search for information"
# Good: "Search Jira issues by JQL query. Supports
#        filtering by project, status, priority,
#        assignee, and date ranges. Returns issue
#        key, summary, status, and priority."
```

Listing 6. Vague vs. specific tool descriptions.

Specific, example-rich descriptions improved routing accuracy by approximately 25%. The most effective descriptions state (1) what data the tool accesses, (2) what query parameters are supported, (3) what fields are returned, and (4) an example use case.

B. Cross-Source Data Conflicts

Occasionally, sources provide conflicting information (e.g., Jira shows 12 open blockers but the analytics dashboard shows 15). The discrepancy typically arises from different query filters, caching latency, or differing definitions of "blocker." The agent currently presents both figures with their sources, allowing the user to reconcile. Automated conflict detection is future work (Section XI).

C. Authentication Complexity

Each tool server requires its own authentication. Managing 14 sets of credentials is operationally complex. We mitigate this with a centralized secrets manager and per-server environment variables, but a unified authentication layer (service mesh, API gateway) would further simplify deployment and credential rotation.

D. Proxy Pattern for Analytics

Three analytics servers (Analytics, Health, Build) are implemented as transparent proxies that forward calls to upstream APIs without modification (Listing 7).

```
# Proxy: forwards calls to upstream unmodified
UPSTREAM_URL = os.environ["ANALYTICS_UPSTREAM"]

@mcp.tool()
async def query_metrics(params):
    response = await httpx.post(UPSTREAM_URL,
json=params)
    return response.json()
```

Listing 7. Thin MCP proxy over an existing analytics API.

This pattern lets the agent access existing analytics APIs with only a thin MCP wrapper—valuable where modifying the upstream service is not feasible. The proxy adds less than 50 ms of latency overhead.

E. Generalizability

Our evaluation is situated in a single large enterprise, and some specifics (the exact 14 sources, port assignments, and the RSD document format) are organization-dependent. The architecture itself, however, is portable: nothing in the discovery, routing, parallel-execution, summarization, traceability, or write-gate mechanisms assumes organization-specific semantics. Adapting the system to another organization requires only re-implementing the per-source MCP servers against that organization's platforms; the agent core is unchanged. The same pattern applies beyond enterprise settings—for instance, a research assistant spanning literature databases, code repositories, and experiment trackers—wherever the task is synthesis across heterogeneous, live, API-accessible sources. We expect the qualitative findings (parallelism is essential for latency, summarization beats truncation, traceability is achievable via prompting plus annotations) to transfer, while the precise quantitative figures will vary with source mix and query distribution.

XI. FUTURE SCOPE

We outline six directions, ordered by estimated impact and feasibility.

A. Real-Time Source Monitoring and Push Alerts

The current architecture is pull-based. A natural extension adds push-based monitoring, where the agent subscribes to change feeds and proactively alerts users when relevant conditions change (e.g., a new P1 issue, or a crash-rate regression above threshold). This requires webhook/WebSocket listeners per server and an event-routing layer, with LLM-based relevance filtering so that not every update triggers an alert.

B. Cross-Source Entity Resolution

Sources reference the same entities with different identifiers. A dedicated entity-resolution layer would maintain a knowledge graph of mappings (Jira project = GitHub org = Health product = analytics suite), bootstrapped from integration metadata and refined through LLM-assisted matching, so the agent can automatically enrich a query across sources without user-specified mappings.

C. Automated Conflict Detection Between Sources

Building on Section X-B, the system would compare structured fields across results, flag discrepancies with a likely cause, and issue targeted follow-up queries (e.g., re-running a Jira search with filters identical to the analytics query) to resolve them.

D. User-Defined Source Priority Weighting

Different roles trust different sources differently. Per-user source-priority weights would influence tool-selection ordering, conflict resolution, and report emphasis, via per-source weight vectors in the user profile and corresponding system-prompt conditioning.

E. Federated Tool Marketplace

As the ecosystem grows, a marketplace would let teams publish their own MCP tool servers with standardized metadata, a review process, dynamic tool-budget management, and usage analytics—paralleling package managers and API marketplaces, adapted for MCP.

F. Semantic Caching of Frequent Queries

Many queries are variations of the same information need. Semantic caching would recognize a near-duplicate query and serve a cached result with a freshness indicator, at two levels: per-tool result caching with per-source TTLs, and synthesized-response caching keyed by query embedding. The key challenge is invalidation; combining push-based monitoring with cache invalidation would yield a system that is both fast and fresh.

XII. CONCLUSION

We presented an architecture for multi-source information synthesis using tool-augmented LLM agents, integrating 14 heterogeneous enterprise data sources through the Model Context Protocol. Multi-source synthesis produces significantly more comprehensive and actionable responses than single-source queries (+40% completeness, +33% actionability), while source traceability ensures 96% of claims in generated reports link back to verifiable origins.

Key insights from our production deployment: (1) **MCP provides the right abstraction**—each source needs a dedicated server, but the protocol standardizes discovery and invocation; (2) **parallel execution is essential** for interactive latency (3–4s vs. 12–18s); (3) **LLM-based summarization outperforms truncation**, retaining 94% of information at 7% of the token cost; (4) **source traceability is achievable** via inline citations plus document-level annotations, reaching 96% attribution without architectural complexity; and (5) **write gates balance capability with safety** and have prevented all categories of accidental writes in production. The architecture is extensible along new sources, new synthesis capabilities, and new output formats.

REFERENCES

[1]. P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *NeurIPS*, 2020.

[2]. Anthropic, "Model Context Protocol Specification," 2025. [Online]. Available: https://modelcontextprotocol.io

[3]. OpenAI, "Function Calling and Other API Updates," 2023.

[4]. Anthropic, "Tool Use with Claude," 2024.

[5]. T. Schick et al., "Toolformer: Language Models Can Teach Themselves to Use Tools," *NeurIPS*, 2023.

[6]. Y. Qin et al., "ToolLLM: Facilitating LLMs to Master 16000+ Real-World APIs," *ICLR*, 2024.

[7]. S. G. Patil et al., "Gorilla: Large Language Model Connected with Massive APIs," *arXiv:2305.15334*, 2023.

[8]. L. Gao et al., "Retrieval-Augmented Generation for LLMs: A Survey," *arXiv:2312.10997*, 2024.

[9]. Google DeepMind, "Agent-to-Agent Protocol," 2025. [Online]. Available: https://a2aprotocol.ai

[10]. Y. Shen et al., "HuggingGPT: Solving AI Tasks with ChatGPT and its Friends," *NeurIPS*, 2024.

[11]. H. Chase, "LangChain: Building Applications with LLMs through Composability," 2023.

[12]. S. Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models," *ICLR*, 2023.

[13]. APPENDIX A: SURVEY INSTRUMENT

[14]. The survey referenced in Section II-D was administered to 85 knowledge workers (44 engineering, 21 program management, 20 product management) via an internal form. Respondents answered with respect to their most recent multi-platform research task. Table A-I maps each survey question to the metric it produced in Table I.

[15]. **Role & tenure (screener).** "What is your primary role?" [Engineering / Program Management / Product Management / Other] and "How long have you been in this role?" [<1y / 1–3y / 3–5y / 5y+].

[16]. **Platforms per task.** "In your most recent research task that required looking across systems, how many distinct platforms did you consult?" [integer].

[17]. **Time per task.** "Approximately how many minutes did that task take, end to end?" [integer minutes].

[18]. **Incomplete analysis.** "Did time pressure cause you to submit an analysis you knew was incomplete?" [Yes / No], with optional free text.

[19]. **Stale data.** "In the last month, have you made or contributed to a decision using data you later learned was out of date?" [Yes / No].

[20]. **Weekly synthesis hours.** "In a typical week, how many hours do you spend gathering and reconciling information across platforms?" [integer hours].

[21]. **Most-used platforms.** "Which platforms do you consult most often for research tasks?" [multi-select: Jira, Confluence, GitHub, Analytics, Community, Build system, Web, Other].

[22]. **Primary pain point.** "What is the single biggest difficulty in cross-platform research?" [context switching / query syntax / correlating results / access & auth / data freshness / other].

[23]. **Confidence (Likert).** "I am confident my cross-platform analyses are complete." [1 = strongly disagree … 5 = strongly agree].

[24]. **Automation value (Likert).** "An assistant that queried all these sources at once and cited its sources would meaningfully improve my work." [1–5].

Table A-I. Survey question → reported metric mapping.

Metric in Table I	Derived from
Avg. platforms consulted per task	Q2 (mean)
Avg. time per multi-platform research	Q3 (mean)
% reporting incomplete analysis	Q4 (% "Yes")
% reporting decisions on stale data	Q5 (% "Yes")
Estimated weekly synthesis hours	Q6 (mean)

Q1 provides the role segmentation (columns of Table I); Q7–Q10 inform the qualitative discussion in Sections II and IV.

Appendix B: Representative Production Queries

To ground the usage analysis in verifiable data, this appendix reports a fully instrumented slice of production traffic captured by the request-logging subsystem (the weekly logs read by the tracing viewer). The slice covers **2026-06-25 to 2026-07-01** and contains **209 top-level user-initiated requests** spanning **162 distinct queries** (parent requests only; internal logging-agent rows excluded). Tool and server attribution is parsed from each request's tool-call events. Requests average **5.75 tool calls** overall (10.64 among tool-using requests) and touch 1.12 distinct servers overall (2.07 among tool-using requests). This one-week instrumented window is a subset of the larger production corpus referenced in Section VIII; its relative source ordering corroborates Table VIII.

All examples are de-identified: e-mail addresses, URLs, Slack channel/user references, user identifiers, @-mentions, and personal name handles have been redacted or replaced with generic placeholders. Table B-I gives the observed server call volume; Table B-II lists representative queries grouped by the intent categories of Table VII, each annotated with the servers it invoked.

Table B-I. Server call volume in the instrumented window (2026-06-25 to 07-01).

Tool server	Calls	Tool server	Calls
jira	460	slack	41
wiki	223	github	25
community	171	github_ent	17
filesystem	116	local	15
docs_search	73	readiness	5
websearch	56		

Top tools: jira/search_issues (297), jira/get_issue (149), wiki/search_content (132), community/search_topics (97), wiki/get_page (89), filesystem/bash (81), websearch/web_search (53). Some servers (e.g. docs_search documentation search, slack, and an internal readiness service) are deployment-specific instances beyond the representative 14 of Table III.

Table B-II. De-identified production queries by intent, with invoked tool servers.

Intent	Representative query (de-identified)	Servers invoked
Release readiness	"Is <product-name> ready for release? Check stability, open blockers, and build status."	local, readiness
	"Can you give a release readiness report for <product-name>?"	readiness
	"Is <product-name> ready for the July release?"	docs_search, jira, wiki
	"What is <product-name> shipping in July?"	jira, wiki
	"Generate production-ready release notes for <product-name> 30.6 (June 2026)."	jira, local, wiki
Bug investigation	"What are the open P1 blockers for the next <product-name> Desktop release?"	jira, wiki
	"What is the status of the Export As feature in <product-name> Desktop? A customer is asking."	community, jira
	"Why would a <product-name> customer see 'Could not update smart object files'?"	community, jira, websearch
	"Users cannot save slices as JPEG using Save for Web—what's going on?"	community, jira
	"Find <product-name> bugs reported publicly since the June release."	community, jira
Feature analysis	"What are customers saying about <product-name> performance on the community forum?"	community
	"Who owns the service pods for <product-name>?"	docs_search, slack, websearch, wiki
	"What are the top community feature requests and pain points for <product-name> Mobile?"	community, local
	"Generate a report on what customers are saying about the <product-name> mobile June release."	community, filesystem, websearch
Code / build review	"Review PR <link>." / "Can you review a <product-name> branch PR?"	github, github_ent
	"Get me the git diff between the two most recent nightly builds."	local
	"A new CI/build event was posted in a monitored thread." (automated posting)	— (notify only)

Intent	Representative query (de-identified)	Servers invoked
General research	"How do I connect Claude to SharePoint?"	websearch
	"Find documentation on configuring Charles to view app instrumentation from <product-name>."	docs_search, wiki
	"Summarize a teammate's daily standup work over the past two weeks."	filesystem, slack
	"What type of short-form videos would <product-name> customers like to create?"	community, docs_search, wiki

Verbatim (de-identified) production queries from the instrumented window. The automated CI/build monitored-thread postings are the single largest driver of the code-review bucket; most collapse into a few distinct templates and invoke no tools (notification only).