Enterprise Agentic AI Should Use Real-Time Discovery & Self-Coding (RTDC) Integration Instead Of Static Tool-Registry Protocols like MCP

(Conference Preprint Version)

Andrey Ryabov, Zenera

Stephane H. Maes,  **, Zenera,** shmaes.physics@gmail.com

Ramu Sunkara, Zenera

Abstract

This position paper argues that static tool-registry protocols, most prominently the Model Context Protocol (MCP), are unsuitable as foundational integration architectures for production-grade agentic AI in heterogeneous enterprise environments, and that the field should converge on *Real-Time Discovery and (Self) Coding* (RTDC) Integration as the necessary architectural alternative. MCP was introduced to standardize connections between language models and external systems through a uniform JSON-RPC 2.0 client-server interface. Despite rapid adoption in developer tooling, we demonstrate that MCP doubles the enterprise integration maintenance and vulnerability surfaces, fails categorically against legacy systems that harbor the most critical enterprise data, exhausts language model context windows through static tool enumeration at production scale, induces selection collapse in large tool registries, and introduces security vulnerabilities, including tool poisoning, prompt injection, and supply chain compromise. These are fundamentally incompatible with enterprise zero-trust architectures. We survey three alternative paradigms: user-interface automation, real-time context lake architectures, and terminal agents. We show that each resolves only a subset of the enterprise integration challenge. We then argue for RTDC Integration: a paradigm in which autonomous meta-agents dynamically discover system interfaces, and schemas, at runtime, synthesize and validate integration code in sandboxed execution environments, and accumulate versioned, reusable capability artifacts through a continuous discovery-synthesize-verify-promote loop. Unlike static protocol registries, RTDC integrates with legacy mainframes, undocumented APIs, and proprietary systems without prior engineering effort, and enables the eventual decommissioning of those systems through autonomous logic internalization. This is a capability no protocol-based approach can provide.

1 Introduction

The enterprise deployment of agentic AI has reached a critical inflection point. Autonomous agents capable of perceiving enterprise context, executing multi-step reasoning, and interacting with heterogeneous software systems promise to transform business operations at scale. Yet empirical evidence from large-scale enterprise deployments consistently reveals that approximately 95% of enterprise AI pilot projects fail to reach production [1]. This failure rate stems overwhelmingly from integration architecture deficiencies, not from limitations in the underlying language models.

Cite as: Andrey Ryabov, Stephane H. Maes, Ramu Sunkara, (2026), "Enterprise Agentic AI Should Use Real-Time Discovery & Self-Coding (RTDC) Integration Instead Of Static Tool-Registry Protocols like MCP" (Conference Preprint Version), https://shmaes.wordpress.com/wp-content/uploads/2026/05/rtdc_integration_instead_static-repository-tool-protocols-like-mcp_preprint_sm_5_2_26_pub.pdf, May 3, 2026, & the extended version: https://zenodo.org/doi/10.5281/zenodo.20482269, https://shmaes.wordpress.com/2026/05/31/enterprise-agentic-ai-should-use-real-time-discovery-self-coding-rtdc-integration-instead-of-static-tool-registry-protocols-like-mcp/.

To resolve the friction of connecting large language models to enterprise systems, Anthropic released the Model Context Protocol (MCP) in November 2024 [2]. MCP provides an open-source standard built on JSON-RPC 2.0, positioning itself as a universal adapter that eliminates the need for custom per-service integration code by standardizing the interface between any compliant AI client and any compliant tool server. MCP, and analogous static tool-registry protocols should not form the foundational integration architecture of enterprise agentic systems. The structural properties that make MCP appealing in small-scale developer contexts, i.e., a fixed, enumerable tool registry mediated by a standard protocol, are the same properties that make it unsuitable for the heterogeneity, scale, and security requirements of production enterprise environments.

The correct long-term architectural response is Real-Time Discovery and (Self-)Coding (RTDC): a paradigm in which AI agents treat integration as a dynamically acquired cognitive skill, autonomously generating, validating, and accumulating executable integration capabilities at runtime rather than consuming pre-enumerated protocol registries. The resulting agentic AI platform is Application-aware.

This position is grounded in four structural claims, developed in Section 2:

- Static tool registries double the integration maintenance surface without eliminating prior obligations.
- 2. Protocol-based integration is systematically ineffective against legacy systems. Ironically, these are the systems that most urgently requiring AI automation.
- Context window consumption and tool selection accuracy degrade as registry size scales to production requirements.
- The security threat model of MCP is incompatible with enterprise zero-trust architectures.

Sections 3 and 4 evaluate alternative paradigms and characterize the RTDC approach and application-aware AI architecture . Section 5 addresses counterarguments. Section 6 concludes with a research agenda.

2 Structural Limitations of the Model Context Protocol

2.1 Duplication of the Integration Maintenance Surface

MCP's stated premise is simplification through standardization. In production enterprise contexts, the protocol achieves the opposite. Prior to MCP, an enterprise maintained a set S of programmatic interfaces, i.e., REST endpoints, SOAP services, internal APIs, of cardinality N. MCP requires constructing a corresponding MCP server for each target interface, creating a parallel wrapper layer also of cardinality N. The total maintained integration surface grows from N to 2N. Also, new interfaces means new vulnerabilities.

Metric	Pre-MCP	With MCP
Primary interfaces	N REST/SOAP endpoints	N REST/SOAP endpoints
MCP wrapper layer	0	N protocol servers
Total maintained surface	N	2N
Schema synchronization events	Per upstream change	Per upstream change, x2

Table 1: Integration surface before and after MCP adoption. The requirement for API is just not disappearing. It would be wrong to assert or assume that one does not need the API anymore, and could just maintain/expose MCP.

This duplication creates a perpetual schema synchronization liability. When an underlying endpoint undergoes a schema change, the corresponding MCP tool definition must be manually updated, and the agent will fail until this happens. If the tool description lags behind the live API, the language model reasons over a stale capability specification and produces confident hallucinations, incorrect parameter invocations, and subtle data corruption that may require weeks of engineering effort to diagnose [3]. The protocol thus transforms a one-time connection task into a perpetual dual-maintenance obligation.

2.2 Failure Against Legacy Systems

MCP is an API-first protocol: it requires an existing programmatic endpoint to wrap. This structural dependency creates an exclusion boundary precisely where enterprise integration pressure is highest. Production enterprise technology stacks are not composed exclusively of modern, well-documented REST services. Organizations depend on deeply entrenched legacy infrastructure: COBOL mainframes, pre-REST SOAP services, proprietary ERP customizations, and undocumented internal databases accumulating decades of operational history [4]. These systems frequently hold an enterprise's most critical operational data—financial records, supply chain state, regulatory compliance recordsregulatory compliance records, and they are the systems most urgently requiring AI automation.

Because legacy systems lack MCP-compatible interfaces, and because constructing wrapper servers for undocumented, stateful, or terminal-based systems is prohibitively complex, MCP provides no integration path for this critical long tail of enterprise software. Organizations adopting MCP-first architectures experience a structural bifurcation: modern SaaS applications receive standardized AI access while critical legacy systems remain isolated behind manual human operators, or brittle batch processes. This bifurcation is precisely the integration gap that enterprise agentic AI must close to generate material business value.

2.3 Context Window Exhaustion and Selection Collapse

MCP communicates available capabilities to the language model by loading all registered tool names, descriptions, and JSON schemas into the inference context prior to each request. For the small tool registries typical of developer tooling, this is tractable. At enterprise scale, it rapidly becomes a bottleneck. A representative enterprise MCP deployment may register hundreds of tools across dozens of internal services. Tool descriptions alone can consume in excess of 150,000 tokens before the model processes a single user instruction. This *context window tax* imposes three simultaneous penalties: (a) increased per-call inference cost proportional to token count; (b) degraded response latency; and (c) categorical exclusion of smaller, more cost-efficient models whose context windows cannot accommodate the full registry.

Beyond raw token cost, large registries induce *selection collapse*: language models exhibit significantly degraded tool-selection accuracy when presented with a large number of semantically similar tool descriptions [5, 6]. When a model must differentiate among dozens of similarly-named functions across multiple protocol servers, it may hallucinate nonexistent tool calls, select tools with similar signatures but incorrect semantics, or fail to select any tool and fall back to natural language approximation. Empirically, protocol-augmented agents achieve only 11.5–16.7% task success on complex enterprise platforms such as ServiceNow [7], consistent with selection collapse as a primary failure mode rather than a limitation of the underlying language model.

2.4 Security Vulnerabilities Incompatible with Enterprise Zero-Trust Postures

MCP's architectural design introduces a class of security vulnerabilities that are difficult to remediate within the protocol model itself.

MCP servers are executable code artifacts distributed across organizational and third-party boundaries. The protocol currently lacks robust centralized identity management and relies on basic OAuth 2.0 implementations that conflict with modern enterprise zero-trust architectures [8]. Without strict containerized isolation and continuous integrity monitoring, a compromised MCP server becomes an insider threat vector carrying language model-level trust. Organizations enforcing RBAC, data classification, and egress controls cannot rely on MCP's current governance primitives to satisfy compliance requirements in regulated industries.

3 Alternative Integration Paradigms and Their Limitations

The structural limitations of MCP have motivated at least three widely used alternative integration paradigms, sometimes also combined with MCP. We evaluate each and show that none individually resolves the full enterprise integration challenge.

Vulnerability	Mechanism	Enterprise Impact
Tool Poisoning	Adversarial modification of tool definitions post-installation	Agents execute unauthorized commands as ostensibly legitimate operations
Tool Shadowing	Malicious servers register tools with names identical to legitimate services	Enterprise data routed to unauthorized third-party endpoints
Prompt Injection	Adversarial instructions embedded in tool descriptions or user inputs	Language model bypasses safety guardrails; executes unintended actions
Supply Chain Compromise	MCP server packages contain hidden backdoors or exfiltration logic	Credential theft, environment variable exfiltration, lateral network movement

Table 2: MCP security vulnerabilities and enterprise impact.

3.1 User Interface Automation

UI automation agents interact with software through its graphical presentation layer, using computer vision and natural language processing to navigate interfaces as a human operator would, and subsequently infer how to access data. This approach bypasses the API requirement: an agent can automate a COBOL terminal application by reading its screen as readily as a modern web application. The primary advantage is rapid deployment across heterogeneous systems without API engineering. The primary limitation is *architectural depth*: UI agents operate on the presentation layer of enterprise logic, not the execution layer. They can automate what a human can see and click, but cannot access the transactional logic, database schemas, or business rules embedded in application backends. Network sniffing, another risky potential source of vulnerabilities, can be used to infer the associated backend call. These approaches require interacting a run time environment, e.g., demo or staging application, to develop the integration. When the system changes, the work has to be repeated.

Using the (Agentic AI) Strangler Fig pattern [9, 10] as a framing device, UI automation successfully executes Phase 1, i.e., surrounding a legacy system with digital workers that automate end-user tasks. It systematically fails at Phase 2, i.e., replacing and decommissioning the core legacy logic, because the agent's only interface remains the UI, leaving the underlying monolith intact, preventing agentic implementation of the inner processes.

Phase	Objective	UI Automation Capability
Phase 1: Surround and extend	Deploy digital workers to automate end-user tasks	Capable
Phase 2: Replace and decommission	Replace core internal processes; retire legacy system	Incapable: presentation-layer dependency preserves the monolith

Table 3: UI automation capability across Strangler Fig phases.

UI agents are also inherently fragile: not only backend changes, like new schema or application version, but also just minor interface changes, CSS updates, menu reorganization, modal dialog insertions, break agent navigation logic. The dependency on visual layout prevents exploiting programmatic shortcuts unavailable at the presentation layer and creates maintenance costs that scale with application change velocity.

3.2 Real-Time Context Lake Architectures

Context lake-like architectures address the *read-path* integration problem. Traditional data lakes and warehouses operate on batch schedules, delivering stale snapshots to downstream consumers. For autonomous agents operating in tight observe-decide-act loops, stale context produces compounding decision errors [11]. Real-time context engines resolve this by transforming raw data updates into pre-computed materialized views through incremental view maintenance [12]. Agents query

canonical business objects, e.g., unified customer profiles, real-time inventory counts, and receive millisecond-latency responses with transactional consistency.

Dimension	Traditional Data Lake	Real-Time Context Engine
Primary consumer	Human analysts; batch processes	Autonomous agents requiring low-latency decisions
Data freshness	Stale batched snapshots	Live, evaluated at decision time
Query processing	On-demand, computationally heavy	Pre-computed via incremental view maintenance
Consistency model	Eventual	Transactional

Table 4: Comparative properties of data lake and real-time context engine architectures.

However, context lake architectures solve only the read path, and as such can be a viable way to cache data for agents. When an agent must *act*, e.g.. modifying a record in a legacy ERP, triggering a workflow, updating a procurement database, it still requires a programmatic execution interface to the target system. Many context engine platforms delegate write-path execution back to MCP, reintroducing all of the protocol's limitations at the action boundary. An agent with perfect observational fidelity but no execution capability is operationally paralyzed, and, by definition, it is not an agent.

3.3 Terminal Agents and Direct Programmatic Access

Terminal agents represent the most capable currently available alternative to protocol-based integration. Rather than invoking predefined tool registries, a terminal agent is equipped with a shell and persistent filesystem. It writes and executes custom code to interact with target systems directly, emulating the Read-Eval-Print Loop (REPL) workflow of a skilled developer [13]. The StarShell framework [7] provides an empirically rigorous evaluation of this paradigm across enterprise platforms:

Architecture	Modality	Success Rate
Protocol-Augmented (MCP)	MCP tool registry	11.5-16.7% (ServiceNow)
Web Agents	GUI / browser rendering	Higher flexibility; severe computational overhead
Terminal Agents	Direct bash / shell scripting	77.5% overall (Gemini 1.5 Pro)

Table 5: Empirical task success across integration modalities (Bechard et al. [7]).

Terminal agents outperform MCP-based agents by a factor of 4-6x on complex enterprise tasks. The programmatic flexibility to compose arbitrary operations, manipulate data directly, and chain commands without pre-enumeration eliminates most structural limitations of static tool registries.

Nevertheless, terminal agents introduce two additional unresolved problems. First, unrestricted shell access is a substantial security surface: a misdirected agent can execute destructive commands, exfiltrate data, or cause cascading system failures. Enterprise deployments require strict containerized sandboxing that adds significant operational overhead. Second, terminal agents still require *some* programmatic interface to exist at the target system. When a legacy mainframe exposes no API, shell endpoint, or file-based interface, terminal agents cannot automate its processes. We are back to the original integration impasse.

4 RTDC Integration: The Necessary Paradigm

The preceding analysis identifies a consistent structural gap: no current paradigm can autonomously generate integration capabilities for previously unencountered systems, and none can decommission legacy systems by internalizing their core operational logic. We argue that resolving this gap requires treating integration not as protocol adherence or pre-registered tool invocation, but as a *dynamically acquired cognitive skill* executed at runtime.

4.1 Principles of RTDC Integration, And Application-aware Agentic AI

RTDC is an architectural paradigm in which AI agents autonomously generate, validate, and accumulate executable integration artifacts at runtime. RTDC in an Application-aware agentic AI platform is defined by five structural principles:

Principle 1: Integration is executable code, not configuration. Every integration artifact is a full executable, e.g., Python, shell, or composite pipeline—capable of expressing arbitrary logic: conditionals, retries, multi-step transactions, and error recovery that declarative tool schemas cannot represent. Executable code is Turing-complete; declarative schemas are not.

Principle 2: Authentication is infrastructure, not application logic. Credential management, e.g., OAuth 2.0 flows, token refresh, API key vaulting, certificate rotation—is delegated to a dedicated authentication abstraction layer. Integration code requests tokens by provider identifier; the platform handles the full credential lifecycle, preventing credential leakage through the model's context window.

Principle 3: Capability artifacts are versioned with full provenance. Every generated integration artifact is an immutable, versioned object stored in transactional storage with complete lineage: author identity (agent or human), generation timestamp, source documentation reference, test results, and a full diff history. Rollback to any prior version is atomic, providing a complete audit trail.

Principle 4: Discovery is semantic, not enumerative. Agents locate capabilities through natural language semantic search over capability descriptions. Only the matched capability's interface contract enters the model's context window; implementation code never pollutes the inference context. This eliminates the context window tax regardless of the total number of registered capabilities.

Principle 5: The system self-extends through use. When an agent encounters a capability gap, e.g., a system it has not previously integrated, an operation no existing artifact cove, it autonomously generates a new integration artifact, validates it in isolation, and persists it to the shared capability store. The integration surface grows as a direct consequence of operational use, requiring no manual engineering effort.

4.2 The Real-Time Discovery and Coding (RTDC) Loop

The core operational mechanism of RTDC Integration, is the RTDC loop: a continuous four-phase cycle governing autonomous capability generation.

Phase 1: Total Enterprise Introspection. The agent discovers the target system's interface by reading available documentation, probing endpoint patterns, parsing response schemas, and inferring authentication mechanisms from observable system behavior. This phase operates on raw system artifacts, e.g., documentation, schemas, terminal responses, without requiring pre-built connectors. The agent characterizes an unfamiliar system empirically, the way a skilled developer would. Constraints, tribal knowledge, policies, and best practices can also be ingested.

Phase 2: Deterministic Constraint Enforcement. Before code generation proceeds, the proposed integration is evaluated against a constraint graph encoding enterprise governance requirements: data classification policies, RBAC-gated resource access, egress restrictions, and rate limits. Only constraint-satisfying operations advance to code synthesis. This ensures that dynamically generated capabilities remain within the organization's policy envelope.

Phase 3: Autonomous Meta-Agent Orchestration. A meta-agent decomposes the integration requirement into executable sub-tasks, synthesizes integration code, and routes it to a sandboxed execution environment for validation against functional tests and schema assertions. The meta-agent performs cross-agent semantic verification, i.e., checking for prompt contradictions across co-deployed agents,,,,and guarantees workflow termination, treating agent coordination as a first-class engineering problem.

Phase 4: Dynamic Capability Generation. Successfully validated integration artifacts are persisted to versioned storage, semantically indexed, and made available for future retrieval by any agent in the system. When upstream APIs change, affected artifacts are flagged for re-validation rather than requiring manual updates. The capability library self-heals when integrated systems evolve.

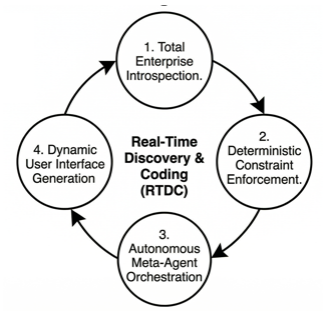

Figure 1: The RTDC Loop, with the four-phase cycle governing autonomous capability generation in Self-Coding Integration. Phases proceed clockwise: Introspection, Constraint Enforcement, Meta-Agent Orchestration, Capability Generation.

4.3 Completing the Strangler Fig Pattern

RTDC integration is the only integration paradigm capable of completing the Strangler Fig pattern in its entirety. By autonomously synthesizing integration code that interfaces directly with legacy system internals, e.g., database schemas, batch processing interfaces, terminal callable interfaces, and proprietary binary protocols, RTDC meta-agents can progressively internalize the business logic of legacy applications. Once this logic is captured in versioned, auditable integration artifacts, the legacy system can be incrementally decommissioned without a disruptive big-bang migration.

This capability distinguishes RTDC integration from all reviewed alternatives: UI agents can surround a legacy system but cannot replace it; context lakes can observe a legacy system but cannot act on it; terminal agents can interact with a legacy system but cannot discover its interfaces autonomously, or replace its inner workflow with smarter ones. RTDC integration can discover, integrate, internalize, and ultimately replace legacy systems through a continuous, governance-compliant process.

Paralyzed by the innovator's dilemma, incumbent vendors of enterprise applications, prefer to expose MCP servers, because this way the core application inner processes are not disrupted. It is exactly what Service Now agentic platform and SalesForce's Agentforce 360 Platform. RTDC integration, and application aware AI platform, address these issues and catalyzes the development of the agentic inner processes, i.e., Phase 2 in table 3.

5 Alternative Views

MCP has legitimate value in constrained contexts. We acknowledge that MCP provides genuine utility for some early-stage development, and for integrating modern SaaS platforms that actively maintain official protocol servers. When an enterprise deploys a greenfield ecosystem with well-documented, stable API surfaces, MCP reduces initial integration development time, and benefits from a growing ecosystem of pre-built servers. Our position is not that MCP should never be used, but that (a) it should not form the *foundational* integration architecture of a production enterprise agentic system, (b) it cannot serve as the primary mechanism for integrating with legacy systems, and (c) organizations that anchor their agent infrastructure to MCP will encounter the structural limitations described in Section 2 as they scale beyond initial pilot deployments.

Self-coding integration introduces novel safety risks. . This is a legitimate concern. RTDC integration agents that generate and execute arbitrary code at runtime represent a significant attack surface if inadequately constrained. We acknowledge that the safety engineering required for production RTDC Integration deployment is non-trivial. Our position is that these risks are *tractable*, i.e., addressable through containerized execution isolation, capability-level RBAC, immutable audit logging, and human-in-the-loop review gates for high-risk capability classes (All, features required of an application-aware agentic AI platform), whereas the structural limitations of MCP identified in Section 2 are not tractable within the protocol model. The empirical 77.5% task success of terminal agents operating under analogous principles [7] demonstrates that the underlying approach is viable; the governance challenge has known solution strategies.

RTDC integration requires more sophisticated infrastructure than MCP. True. RTDC integration requires durable workflow orchestration, transactional storage for versioned artifacts, sandboxed execution environments, and semantic retrieval infrastructure for capability discovery. All this is what we characterize as an application-aware agentic AI platform. For organizations at the earliest stages of AI adoption, this infrastructure investment may appear disproportionate to immediate needs. We acknowledge that MCP and static tool registries can serve as practical on-ramps. Our position concerns the *long-term architectural trajectory*: organizations that do not plan for eventual migration to RTDC integration-class architectures will encounter binding structural constraints as their agentic systems mature. Such application-aware platforms are available, e.g., [16].

6 Conclusions

Enterprise agentic AI systems require an integration architecture commensurate with the complexity, heterogeneity, and operational criticality of real enterprise technology stacks. MCP, despite genuine utility in developer tooling and early-stage SaaS deployments, is structurally unsuitable as a foundational enterprise integration paradigm. It doubles the maintenance surface, excludes legacy systems categorically, exhausts context windows at scale, induces selection collapse, and introduces security vulnerabilities incompatible with enterprise governance requirements. Alternative paradigms, i.e., UI automation, context lake architectures, and terminal agents, and hybrids of these, or with MCP, resolve subsets of this problem space but fail to address the full enterprise integration lifecycle. RTDC Integration, through its RTDC loop of autonomous discovery, constraint-validated synthesis, sandboxed execution, and versioned artifact accumulation, is the only paradigm capable of integrating the full breadth of enterprise systems, including legacy systems with no modern programmatic interfaces, and enabling their eventual decommissioning through autonomous logic internalization. We call on the research community to prioritize: (1) formal characterization of RTDC integration capability correctness and convergence properties under distribution shift in upstream APIs; (2) adversarial robustness of sandboxed code generation environments against prompt injection and escape attacks; (3) governance frameworks for human oversight of autonomously generated integration artifacts in regulated industries; (4) empirical benchmarks spanning the full heterogeneity of production enterprise environments, including legacy mainframe and proprietary protocol coverage; and (5) Application-aware agentic AI framework to ensure enterprise grade agentic AI capabilities built with RTDC integration, and enabling the full agentic AI strangler fig pattern.

References

[1] McKinsey Global Institute. The state of AI in early 2024: Gen AI adoption spikes and starts to generate value. *McKinsey & Company*, 2024.

[2] Anthropic. Model Context Protocol Specification. `https://modelcontextprotocol.io/specification`. 2024.

[3] Chen, M., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[4] Gartner Research. Legacy modernization challenges in enterprise technology stacks. *Gartner*, 2023.

[5] Shi, F., et al. Large language models can be easily distracted by irrelevant context. *Proceedings of the 40th International Conference on Machine Learning, PMLR*, 2023.

[6] Qin, Y., et al. Tool Learning with Foundation Models. *arXiv preprint arXiv:2304.08354*, 2023.

[7] Bechard, P., et al. Terminal Agents Suffice for Enterprise Automation. *arXiv preprint arXiv:2604.00073*, 2026.

[8] Rose, S., et al. Zero Trust Architecture. *NIST Special Publication 800-207, National Institute of Standards and Technology*, 2020.

[9] Fowler, M. StranglerFigApplication. `https://martinfowler.com/bliki/StranglerFigApplication.html`. 2004.

[10] Microsoft Azure Architecture Center. Strangler Fig Pattern. 2026.

[11] Zhu, X., et al. Ghost in the machine: Challenges of agentic AI in high-stakes enterprise workflows. *arXiv preprint*, 2025.

[12] McSherry, F., et al. Differential dataflow. *4th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[13] Wang, X., et al. Executable Code Actions Elicit Better LLM Agents. *Proceedings of the 41st International Conference on Machine Learning, PMLR*, 2024.

[14] Yao, S., et al. ReAct: Synergizing Reasoning and Acting in Language Models. *International Conference on Learning Representations (ICLR)*, 2023.

[15] Schick, T., et al. Toolformer: Language Models Can Teach Themselves to Use Tools. *Advances in Information Processing Systems 36*, 2023.

[16] Zenera: Technology Deep Dive `https://zenera.ai/technology/deep-dive`, 2026.