

Optimizing Agentic Interfaces for Token Efficiency, Semantic Fidelity and Execution Reliability

A Q2 2026 publication by VIA Research

Arnoud Kleinloog

via.vigensis.com

Swiss engineering. AI-native with Human Insight

June 15, 2026

Contents

Theoretical Fundamentals of Token Economics and the Context Window 1
The Model Context Protocol (MCP): Architectural Promise and Empirical Friction ... 2
The Renaissance of Programmatic Capabilities: Code Execution and Skills 4
Semantic Topologies: The Theoretical Case for GraphQL over REST 5
Navigating the Schema Scale Limit: Semantic Introspection 7
Data Serialization and Syntactic Compression Strategies 8
Architectural Synthesis: The Ideal Agentic API Blueprint 10

The rapid evolution of foundation models has catalyzed a fundamental paradigm shift in software architecture, transitioning enterprise systems from human-centric graphical user interfaces and basic conversational retrieval systems to complex, autonomous agentic workflows. As modern coding agents and state-of-the-art foundation models (such as Antropic's Claude family, OpenAI's GPT models and open-source equivalents) are deployed to execute sophisticated, multi-step integrations, the underlying application programming interfaces designed to service them have emerged as the primary bottleneck determining operational viability. The conventional industry consensus has increasingly gravitated towards MCP (Model Context Protocol) as the default standard for exposing tools and backend systems to artificial intelligence. Concurrently, traditional REST endpoints distributing JSON payloads remain ubiquitous.

However, rigorous empirical research, coupled with an analysis of theoretical fundamentals, indicates that these default postures are deeply flawed when evaluated through the critical lens of token efficiency, context window preservation, and semantic fidelity. Optimizing systems for autonomous agents requires a sophisticated understanding of how foundation models process structural data. This optimization involves a nuanced synthesis of semantic data modeling—specifically leveraging GraphQL to maintain sharp semantic domains rather than generic operations—and file-system-based skill discovery that bypasses the architectural redundancy of pervasive server protocols. The subsequent analysis explores the theoretic fundamentals of the token economy, the documented deficiencies of current protocol layers, the validation of programmatic code execution over traditional tool calling, and the optimal design of semantic data layers for agentic consumption.

Theoretical Fundamentals of Token Economics and the Context Window

To comprehend the necessity of optimizing application programming interfaces for state-of-the-art agents, one must first analyze the theoretical economics and operational physics of the large language model context window. The fundamental architecture of transformer-based models relies on self-attention mechanisms, where the computational complexity and memory requirements scale quadratically with the sequence length of the input. Consequently, token efficiency is not merely an economic cost-saving measure; it functions as a strict upper bound on an agent's cognitive and reasoning capacity. When context windows become congested with extraneous structural data, verbose schema definitions, and repetitive system prompts, the effective reasoning horizon of the agent

 © 2026 Vigensis

diminishes proportionally. This dilution leads to degraded performance, heightened hallucination rates, and execution failures across complex operational chains.

The architecture of most modern foundation models requires stateless interactions via application programming interfaces. This statelessness dictates that for every sequential reasoning step an agent takes, the entire state of the ongoing interaction must be perfectly reconstructed and transmitted back to the model.1 This state reconstruction inherently includes the foundational system prompt, the entirety of the active tool definitions and schemas, the accumulated conversation history, any retrieved documents or vector search results, and ongoing execution telemetry.2 In a multi-step autonomous workflow, a system that initially consumes a trivial number of tokens can rapidly experience a cascading context explosion. Research into token consumption patterns reveals that a standard agent performing a mere ten reasoning steps can easily process between fifty thousand and one hundred thousand tokens per task, simply due to the repeated transmission of verbose tool descriptions and intermediate data payloads.2

This compounding mathematical constraint demands a radical architectural shift from the traditional “fetch and parse” paradigm of human-oriented application programming interfaces to an optimized “discover and orchestrate” paradigm for agents. In traditional distributed systems, over-fetching data from a backend server incurs a relatively minor latency penalty over the network. In agentic systems, however, over-fetching results in a compounding token penalty on every subsequent inference step, permanently polluting the context window.3 Therefore, the optimal integration layer must adhere to three fundamental theoretical mandates: it must minimize the structural metadata presented to the model prior to execution, it must eliminate intermediate data round-tripping across the context boundary, and it must maintain a highly precise, tailored semantic map of available domain operations to prevent agent confusion and schema execution retries.

The Model Context Protocol (MCP): Architectural Promise and Empirical Friction

MCP has rapidly emerged as a prominent, highly publicized standard designed to bridge the gap between foundation models and external enterprise data systems. Originally developed to standardize the interoperability of tools, the protocol operates as an open JSON-RPC-based standard that wraps existing application programming interfaces, providing a uniform layer for capability discovery and stateful session management.4 By maintaining a persistent session between the client agent and the host server, the protocol theoretically mitigates the stateless nature of traditional interfaces. During initialization, both the client and the server engage in a mutual handshake, advertising their supported capabilities, resources, and tools, thereby creating a negotiated contract for the session.4

Despite this theoretical promise, as enterprise architects deploy the protocol at scale, significant architectural and economic friction has been observed. A primary critique, echoing concerns regarding infrastructural bloat, is that the protocol introduces an entirely new, redundant integration surface that requires ongoing maintenance.5 Platform engineering teams are currently navigating a fragmented taxonomy of network gates. Modern infrastructure is now frequently divided into the application programming inter-

 © 2026 Vigensis

face gateway—handling traditional routing, rate limiting, and authentication for REST and GraphQL traffic—the artificial intelligence gateway, and the distinct MCP gateway.6 This "three-gate" model is increasingly viewed as an architectural liability, weaponized by point-solution vendors to justify separate products and billing relationships, despite handling fundamentally overlapping traffic paradigms.6 The consensus among critical architects is that direct programmatic interaction with tried-and-proven REST APIs or GraphQL endpoints, facilitated through traditional tool calls, is often vastly superior to managing a fleet of dedicated protocol servers.5

The Pathology of Semantic Smells and Schema Retries

The core vulnerability of MCP, and indeed any system relying on massive injected tool definitions, lies in its reliance on natural-language tool descriptions. Foundation models depend entirely on these textual descriptions to understand a tool's underlying purpose, select the appropriate operation from a crowded namespace, and map the correct arguments to the execution payload.7

Empirical research evaluating the ecosystem highlights severe, systemic deficiencies in this area. A comprehensive academic study analyzing eight hundred and fifty-six tools spread across one hundred and three discrete servers discovered that an overwhelming majority of tool descriptions contain structural or semantic defects, formally classified as code or specification "smells".7 The statistical prevalence of these defects is staggering, with researchers finding that 97% of the analyzed tool descriptions contained at least one identifiable smell.7 Most critically, 56% of the tools failed to articulate their core purpose clearly to the agent.7

These semantic smells directly contribute to execution retries, schema mapping mistakes, and hallucinated arguments, destroying the token efficiency of the workflow. While researchers attempted to augment these descriptions to adhere to strict quality rubrics, the results presented a difficult trade-off. Comprehensive augmentation improved the model's task success rate by a median of 5 percentage points and improved partial goal completion by 15%; however, it simultaneously inflated the token count, increasing the number of execution steps by 67% and actively regressing performance in 16% of cases.7 The research definitively concludes that achieving performance gains through verbose protocol descriptions is highly inefficient, necessitating the development of highly compact variants that preserve behavioral reliability while excising unnecessary token overhead.7

The Context Dilution Crisis of Static Loading

The token consumption crisis is most acutely evident in the protocol's static loading mechanisms. Standard protocol clients load all available tool definitions upfront, directly into the agent's context window, exposing them to the model using direct tool-calling syntax.10 When an architecture scales beyond simple prototypes to include dozens of enterprise servers exposing hundreds of capabilities, the context window is immediately flooded with schemas before any actual reasoning or user interaction begins.12

This static loading mechanism creates an artificial ceiling on agent capabilities. The resulting reduction in effective context length forces more frequent context compactions,

 © 2026 Vigensis

imposes hard limits on simultaneous server connections, and exponentially drives up inference application programming interface costs.12

Capability Loading Mechanism	Tool Volume	Initial Context Token Consumption	Architectural Impact
Static Loading (Standard Protocol)	60 tools	~47,000 tokens	Severe context dilution, high baseline cost
Dynamic Discovery / File Scanning	60 tools	~400 tokens	99% token reduction, highly optimized

The empirical data demonstrates that static loading of interface schemas is mathematically unsustainable for advanced autonomous agents that require access to thousands of tools.12 To resolve this, architectures must shift toward dynamic tool discovery, keeping the vast majority of capabilities completely out of the active context until explicitly required by the agent's current deterministic reasoning step.12

The Renaissance of Programmatic Capabilities: Code Execution and Skills

To address the profound token bloat associated with verbose schema injection and dedicated protocol servers, a radical architectural shift is underway: representing capabilities not as injected interface definitions, but as executable code discovered dynamically via a local file system. This methodology, often referred to as "Code Execution" or the utilization of "Skills," fundamentally alters how agents interact with external systems, directly aligning with the capabilities of state-of-the-art coding agents.10

Instead of pushing all available schemas into the prompt, the system relies on the foundation model's inherent ability to write and execute code, leveraging the local file system or a sandboxed environment as the primary discovery mechanism.10 Under this paradigm, operations are defined as discrete programmatic files, such as highly typed TypeScript modules with strict interfaces defining input and response objects.11 Because these definitions exist statically on a disk rather than residing in the active prompt context, they incur zero token cost until they are actively inspected by the agent.11

An agent tasked with a specific goal can navigate the directory structure, read the relevant type definitions on demand, and write a transient script to orchestrate the necessary application programming interface calls.11 This methodology perfectly mirrors the concept of skill-based routing, where capabilities are represented by simple markdown files and execution scripts. The agent's harness can quickly scan the frontmatter metadata of available skills, consuming only a few dozen tokens, and only pull the full instructions into context if the current task dictates its necessity.11

Eliminating Intermediate Data Round-Tripping

The most significant theoretical advantage of programmatic code execution over standard interface protocols is the complete elimination of intermediate data round-tripping.11 In a standard tool-calling loop, an agent invokes a tool, the system executes the request, and the raw output is unconditionally appended to the context window.10

 © 2026 Vigensis

This forces the model to ingest potentially massive payloads of intermediate data that it does not actually need to read.

For example, consider an agent instructed to download a meeting transcript from a cloud storage repository and attach it to a specific customer relationship management record.10 Using standard tool protocols, the agent calls the storage tool, the entire transcript is downloaded and injected into the agent's context window, consuming thousands of tokens, and then the agent must pass that entire transcript back out in a subsequent tool call to the customer relationship system.10 This "round-tripping" introduces massive latency, explodes token costs, and creates significant opportunities for the language model to hallucinate or corrupt the data during transmission.11

Conversely, programmatic execution allows the agent to write a simple script that pipes the data directly between the two external systems without the data ever crossing into the foundation model's context boundary.11 The agent generates code to extract the document and pass it directly to the update function. The intermediate data never touches the context window, drastically reducing token consumption, lowering latency by avoiding sequential inference steps, and mitigating severe data privacy risks by preventing sensitive transcripts from being processed by the language model.10

Research implementations of this exact programmatic paradigm have demonstrated breathtaking efficiency gains. In specific benchmarks evaluating complex, multi-tool workflows, shifting from static protocol schemas to dynamic code execution reduced the required context overhead from one hundred and fifty thousand tokens down to a mere two thousand tokens—realizing a time and cost savings of 98%.10 These empirical results confirm the initial hypothesis: modern coding agents are perfectly well able to handle programmatic interfaces natively, and exposing them via skills rather than persistent protocol servers is vastly more efficient.11

Semantic Topologies: The Theoretical Case for GraphQL over REST

While file-based skill discovery and code execution optimize the orchestration and capability routing layer, the underlying structural data layer must also be fundamentally redesigned for agentic consumption. The industry default of exposing Representational State Transfer (REST) endpoints or generic application programming interfaces frequently fails to provide the semantic precision required by autonomous agents. This deficiency is rooted in the theoretical underpinnings of RESTful architectures, which rely on generic HTTP verbs (GET, POST, PUT, DELETE) interacting with static, server-defined resource shapes.3

When an agent queries a standard REST endpoint, it is immediately subjected to systemic over-fetching. The server returns a comprehensive, monolithic object representation replete with creation timestamps, deeply nested database identifiers, relational links, and extraneous metadata that is entirely irrelevant to the agent's immediate reasoning task.3 This unstructured data dump pollutes the context window and forces the foundation model to expend scarce cognitive capacity filtering the signal from the noise.3

 © 2026 Vigensis

Furthermore, flat REST architectures suffer profoundly from the "n+1" problem, which becomes catastrophic in stateless agent loops.16 Enterprise content models and application states are rarely flat; they consist of deeply nested relationships.16 A standard REST architecture requires the agent to execute multiple, sequential tool calls to resolve these relationships across different endpoints. This under-fetching leads to increased network overhead and latency, but more critically, it incurs the repeated token overhead of tool invocation headers, system prompts, and execution metadata with every single loop.3

GraphQL presents a vastly superior theoretical model for agentic interaction by shifting the locus of control from the server entirely to the client agent. Designed around a strictly typed, introspective graph schema, GraphQL completely eliminates over-fetching by allowing the agent to define highly precise selection sets.3 An agent can formulate a tailored query to extract specifically the isolated data points required for its current task—requesting merely a user's name rather than their entire account object—thereby entirely bypassing the token penalty associated with predetermined REST response shapes.3

Beyond output filtering, GraphQL enables single-request orchestration.3 A highly capable coding agent can traverse an entire graph of related services, gathering diverse datasets from inventory, user profiles, and transaction histories in a single, optimized execution.3 This avoids the latency, hallucination risk, and context bloat of sequential tool loops.3 Human developers have long preferred GraphQL over JSON/REST for its ability to keep semantics sharper; for foundation models, this semantic sharpness is not a convenience, but a prerequisite for reliable autonomy. It shifts the interaction from generic resource manipulation to highly specific, semantic domain operations.3

Empirical Validation: Sequential Function Calling and GraphQLRestBench

The theoretical assertion that GraphQL schemas provide superior semantic mapping for agents has been rigorously validated through academic and enterprise research. Researchers at IBM have extensively investigated the specific challenges large language models face when attempting sequential function calling using interdependent variables across traditional REST application programming interfaces.17 The core issue is that models struggle to maintain the variable mapping and response structural dependencies when executing complex chains of generic REST calls.

To address and evaluate this, the researchers developed the GraphQLRestBench dataset, a custom framework consisting of natural language utterances paired with complex sequence mappings of real-world operations across domains like Spotify and the Movie Database.17 The critical innovation of this research was substituting the standard REST response structure representations in the model's prompt with the GraphQL schema of those same interfaces.17

By using the GraphQL schema as a highly reliable, strongly typed source of truth regarding the response structure, the foundation models demonstrated a significantly improved capacity to accurately reason through sequential function calls.17 The Abstract Syntax Tree nature of GraphQL queries aligns perfectly with the code-generation strengths of modern models, providing a deterministic, machine-readable map of the domain that generic REST OpenAPI specifications simply cannot match.16 The strong typing inherent

 © 2026 Vigensis

in GraphQL essentially acts as a guardrail, drastically reducing the instances of schema mistakes and the subsequent need for token-intensive execution retries.16

Server-Side Token Optimization Techniques

Modern GraphQL server implementations, specifically designed for artificial intelligence, provide sophisticated mechanisms to minimize the surface area exposed to the model, further optimizing token efficiency. Enterprise-grade implementations, such as the Apollo ecosystem, allow developers to utilize concise inline comments directly within the operations rather than exposing verbose, autogenerated schema documentation.3 These precise comments guide the agent on formatting expectations, significantly reducing the size of the metadata pushed into the context window.3

Furthermore, developers can utilize hard-coded arguments to remove static parameters from the agent's view entirely, preventing the model from hallucinating invalid inputs.3 Aliases can be employed aggressively to translate obscure, legacy backend nomenclature into semantically clear, concise, domain-specific terminology that the artificial intelligence can easily interpret, saving tokens on both the input schema and the output generation.3 By moving deterministic execution and complex policy enforcement into the graph abstraction layer, the agent is freed to focus purely on high-level reasoning and capability routing, unburdened by underlying system complexity.3

Navigating the Schema Scale Limit: Semantic Introspection

Despite the profound architectural advantages of GraphQL, deploying it natively in agentic ecosystems introduces a critical scaling challenge known as the introspection bottleneck. The standard GraphQL introspection query, which allows clients to discover the available types, queries, and mutations, returns the entirety of the graph.20 For enterprise applications, these schemas are exceptionally large, often encompassing thousands of fields and interconnected types. Forcing an autonomous agent to ingest a massive, monolithic schema on every operational turn, or expecting it to store the schema locally and run sequential grep commands over it, risks immediate context exhaustion and incurs exorbitant financial costs.20 While standard GraphQL schemas are mathematically more concise than OpenAPI specifications, they still suffer from this scaling limitation.20

This critical limitation has catalyzed the development of "Semantic Introspection," a revolutionary extension to the GraphQL specification designed explicitly by artificial intelligence working groups to serve foundation models.20 Rather than traversing the entire schema graph via standard introspection, semantic introspection introduces a standardized __search capability that allows agents to discover relevant operations using purely natural language queries.20

The mechanics of this specification rely on an active vector, embedding, or BM25 index maintained by the GraphQL server, which continuously indexes type names, field names, and semantic descriptions.20 When an agent requires a capability—for instance, identifying how to search for a user by email—it issues a search query describing its intent. The server processes this request and returns a highly targeted result set (__SearchResult)

 © 2026 Vigensis

containing only the most relevant schema coordinates, the matched definitions, and the exact paths from the root query to the necessary definition (pathsToRoot).20

This "discover-then-resolve" workflow allows the agent to extract the precise structural coordinates required to formulate its operation without ever viewing the unrelated, sprawling portions of the enterprise API.20 The pagination model allows the agent to fast-forward through results using an opaque cursor, guaranteeing that token consumption remains strictly linear and bound to the agent's immediate needs.20

Quantitative analysis demonstrates the profound token efficiency of this semantic discovery approach. When benchmarking discovery methodologies for foundation models interacting with a standard application programming interface, the differences in token consumption represent orders of magnitude in operational cost and context preservation.20

Application Discovery Methodology	Context Tokens Consumed	Cost Impact (USD)	Architectural Assessment
OpenAPI Specification (REST)	665,564 tokens	$0.3950	Highest context pollution, structurally verbose
Standard GraphQL Introspection	133,441 tokens	$0.1072	High token density, schema scale problem
GraphQL with Semantic Introspection	59,067 tokens	$0.0895	Highly optimized, precise targeted discovery

By implementing semantic introspection, organizations can achieve a fraction of the cost associated with standard OpenAPI discovery models, cutting token usage by over 90% compared to raw REST documentation.20 The semantic hints and paths to the root provided by the search results enable agents to intelligently navigate complex schemas, dramatically improving the accuracy of generated operations while entirely eliminating the reliance on heuristic, purely text-based schema exploration.23

Data Serialization and Syntactic Compression Strategies

Beyond the structural design and discovery mechanisms of the application programming interface, the raw serialization format used to transport data between the backend server and the foundation model plays a surprisingly critical role in token optimization. While JavaScript Object Notation (JSON) remains the ubiquitous, uncontested standard for modern web applications, it is uniquely poorly suited for large language model interactions due to its immense syntactic overhead.25

The strict syntax requirements of JSON—specifically the necessity for opening and closing tags, redundant curly braces, rigid array brackets, and pervasive quotation marks around every string and key—generate excessive character counts that translate directly into wasted tokens.26 Furthermore, foundation models frequently struggle with the syntax sensitivity of the format. A single missed comma can invalidate a payload, causing the agent to stall or trigger an execution retry.26 Models also demonstrate a tendency to hallucinate unnecessary prefix or suffix text when generating strict JSON outputs, further bleeding tokens and complicating the parsing logic of the surrounding orchestrator.26

 © 2026 Vigensis

Theoretical optimizations in advanced prompt engineering and agent architecture dictate a shift toward alternative serialization formats specifically tailored for machine-learning context windows. YAML (YAML Ain't Markup Language) provides an immediate architectural upgrade over JSON. By replacing brackets and quotation marks with strict indentation, YAML preserves the complex structural hierarchy of nested objects while significantly reducing the token density required to express identical semantic information.26 Empirical benchmarking demonstrates that simply converting a standard data payload from JSON to YAML yields a direct token reduction of between 10 and 25%, simultaneously enhancing the readability of the context window for debugging and accelerating the inference generation time.25

However, for specific data topologies, even greater efficiencies can be realized by abandoning traditional nested structures entirely and adopting highly specialized token-compression formats, such as Token Oriented Object Notation (TOON) or structurally lean tabular variants (LEAN).25 These formats excel spectacularly when processing flat, uniform data arrays, such as lists of database records, numerical telemetry, or tabular search returns. By defining the data schema precisely once at the header and subsequently listing rows via simple delimiters without repeating field names, quotes, or structural braces, these formats strip away all redundant structural metadata.25

Serialization Format	Architectural Suitability & Data Shape	Benchmark Average Tokens	Token Savings vs JSON Baseline	Accuracy (Instruction Adherence)
JSON (Baseline)	Deeply nested, irregular legacy payloads	7,401 tokens	0.0% (Baseline)	86.2%
YAML	Human-readable config, moderate nesting	5,647 tokens	~23.7%	87.4%
Specialized (LEAN)	Flat, tabular data, uniform record arrays	3,939 tokens	~46.8%	87.9%

The empirical benchmark data clearly indicates that utilizing lean, tabular formats for uniform datasets can reduce token consumption by nearly half (46%) compared to the JSON baseline.28 Crucially, this massive reduction in token expenditure does not degrade the model's reasoning accuracy. In fact, removing the extraneous JSON syntax frequently improves instruction adherence and data extraction accuracy (rising from 86.2% to 87.9%), as the model's attention heads are not distracted by processing repetitive structural noise.28

It is important to note, however, that as data complexity increases and hierarchical nesting deepens with mixed types, the theoretical advantage of tabular formats diminishes rapidly. Deeply nested TOON or CSV formats become harder for the model to read and maintain than their JSON counterparts.25 Therefore, production systems must adopt a dynamic, content-aware serialization strategy: utilizing lightweight tabular encoders strictly for flat, list-based query returns, while gracefully falling back to YAML (or optimized tools like BAML) for deeply nested graph traversals and configuration generation.25 This ensures the agent benefits from maximum compression without sacrificing the necessary structural clarity required to comprehend complex object relationships.

 © 2026 Vigensis

Architectural Synthesis: The Ideal Agentic API Blueprint

Synthesizing the theoretical fundamentals of the context window, empirical benchmarking of serialization protocols, and the semantic modeling techniques of modern schemas yields a definitive, highly optimized architectural blueprint for designing application programming interfaces specifically for state-of-the-art coding agents. The optimal integration layer must completely abandon the monolithic, stateless characteristics and generic verbs of traditional web development in favor of a highly modular, semantic, and token-aware design.

First, the core enterprise data layer must be exposed through a strongly typed, semantic graph schema, explicitly rejecting the flat, generic Representational State Transfer endpoints that inherently cause token-destroying over-fetching.3 The GraphQL paradigm provides the exact mathematical structure and tailored querying required for foundation models to reason about complex data relationships deterministically, preventing schema mistakes and the resulting execution retries.3 The exposed schema itself must be tightly controlled by the platform engineers, using aliases, hard-coded variables, and concise comments to present the most minimal, token-efficient map of domain capabilities possible.3

Second, to solve the scale problem inherent in exposing large enterprise schemas to stateless agents, the architecture must implement the Semantic Introspection protocol.20 Autonomous agents should never be forced to parse comprehensive documentation, massive OpenAPI specifications, or full standard introspection graphs. Instead, the backend server must provide an indexed, natural-language search endpoint that delivers localized, highly relevant schema coordinates and paths-to-root on demand.20 This targeted discovery mechanism ensures that the agent's context window remains pristine, populated only by the specific semantic definitions required for its immediate, localized task.20

Third, the orchestration and capability execution layer should heavily leverage the programmatic Code Execution and "Skills" paradigm rather than relying on heavy, persistent MCP servers.10 Rather than utilizing conversational, JSON-based tool-calling loops that pass massive, intermediate data payloads back through the agent's context window, the system should empower the coding agent to write transient programmatic scripts that pipe data directly between services on the host machine.10 By representing integration capabilities as local, discoverable TypeScript or Python files on a file system, the initial token loading cost drops to absolute zero, data privacy is preserved, and execution reliability increases as complex data transformations are offloaded to a deterministic runtime environment rather than the probabilistic language model.11

Finally, the transport serialization layer must abandon default JSON encoding for all interactions directly involving the language model's context window.25 Responses generated by the semantic graph layer should be dynamically encoded into YAML for nested object structures, or lean tabular formats for flat arrays, prior to being injected into the agent's prompt harness.25

By adhering to this comprehensive, multi-tiered optimization strategy—combining semantic GraphQL domains, dynamic search-based introspection, programmatic code ex-

 © 2026 Vigensis

ecution, and syntactic serialization compression—architects can build robust integration ecosystems. This blueprint maximizes the effective reasoning horizon of state-of-the-art models, drastically lowers ongoing inference and architectural maintenance costs, and fundamentally elevates the reliability, speed, and autonomy of the entire agentic system.

Works cited

1. Reducing token costs on autonomous LLM agents - how do you deal with it? : r/LLMDevs, accessed June 14, 2026, https://www.reddit.com/r/LLMDevs/comments/1qnuyyj/reducing_token_costs_on_autonomous_llm_agents_how/
2. How I Reduced LLM Token Costs by 90% Building AI Agents With OpenAI and Claude, accessed June 14, 2026, https://medium.com/@ravityuval/how-i-reduced-llm-token-costs-by-90-using-prompt-rag-and-ai-agent-optimization-f64bd1b56d9f
3. Every Token Counts: Building Efficient AI Agents with GraphQL and ..., accessed June 14, 2026, https://www.apollographql.com/blog/building-efficient-ai-agents-with-graphql-and-apollo-mcp-server
4. MCP vs. REST: What's the right way to connect AI agents to your API? - WorkOS, accessed June 14, 2026, https://workos.com/blog/mcp-vs-rest
5. Choosing between APIs, MCP, and Agent-to-Agent architectures | by Lak Lakshmanan, accessed June 14, 2026, https://lakshmanok.medium.com/choosing-between-apis-mcp-and-agent-to-agent-architectures-b88310e87733
6. The Three Gates of AI Infrastructure: API, AI, and MCP Gateways Compared - Zuplo, accessed June 14, 2026, https://zuplo.com/learning-center/three-gates-ai-infrastructure-api-ai-mcp-gateway
7. [2602.14878] Model Context Protocol (MCP) Tool Descriptions Are Smelly! Towards Improving AI Agent Efficiency with Augmented MCP Tool Descriptions - arXiv, accessed June 14, 2026, https://arxiv.org/abs/2602.14878
8. Model Context Protocol (MCP) Tool Descriptions Are Smelly! Towards Improving AI Agent Efficiency with Augmented MCP Tool Descriptions - arXiv, accessed June 14, 2026, https://arxiv.org/html/2602.14878v1
9. Model Context Protocol (MCP) Tool Descriptions Are Smelly! Towards Improving AI Agent Efficiency with Augmented MCP Tool Descriptions | Request PDF - ResearchGate, accessed June 14, 2026, https://www.researchgate.net/publication/400856195_Model_Context_Protocol_MCP_Tool_Descriptions_Are_Smelly_Towards_Improving_AI_Agent_Efficiency_with_Augmented_MCP_Tool_Descriptions
10. Code execution with MCP: building more efficient AI agents - Anthropic, accessed June 14, 2026, https://www.anthropic.com/engineering/code-execution-with-mcp
11. Code execution with MCP: Building more efficient agents, accessed June 14, 2026, https://simonwillison.net/2025/Nov/4/code-execution-with-mcp/
12. MCP vs mcp-cli: Dynamic Tool Discovery for Token-Efficient AI ..., accessed June 14, 2026, https://techcommunity.microsoft.com/blog/azuredevcommunityblog/mcp-vs-mcp-cli-dynamic-tool-discovery-for-token-efficient-ai-agents/4494272

 © 2026 Vigensis

13. Introducing advanced tool use on the Claude Developer Platform - Anthropic, accessed June 14, 2026, https://www.anthropic.com/engineering/advanced-tool-use
14. Scaling Agents with Code Execution and the Model Context Protocol | by Madhur Prashant, accessed June 14, 2026, https://medium.com/@madhur.prashant7/scaling-agents-with-code-execution-and-the-model-context-protocol-a4c263fa7f61
15. GraphQL: The Superior Choice for LLM-Based Applications | TechConative AI, accessed June 14, 2026, https://techconative.ai/blog/graphql-the-superior-choice-for-llm-based-applications
16. GraphQL vs. REST at enterprise scale: Performance benchmarks - Contentstack, accessed June 14, 2026, https://www.contentstack.com/blog/tech-talk/graphql-vs-rest-at-enterprise-scale-performance-benchmarks
17. Sequential API Function Calling Using GraphQL Schema for EMNLP 2024 - IBM Research, accessed June 14, 2026, https://research.ibm.com/publications/sequential-api-function-calling-using-graphql-schema
18. Sequential API Function Calling Using GraphQL Schema - ACL Anthology, accessed June 14, 2026, https://aclanthology.org/2024.emnlp-main.1083.pdf
19. Sequential API Function Calling Using GraphQL Schema - ACL Anthology, accessed June 14, 2026, https://aclanthology.org/2024.emnlp-main.1083/
20. Semantic Introspection - ChilliCream GraphQL Platform, accessed June 14, 2026, https://chillicream.com/blog/2026/04/22/semantic-introspection/
21. What's new in Fusion 16 - ChilliCream GraphQL Platform, accessed June 14, 2026, https://chillicream.com/blog/2026/05/15/fusion-16/
22. Semantic Introspection - Pascal Senn, ChilliCream - YouTube, accessed June 14, 2026, https://www.youtube.com/watch?v=2Ud1gxUpgrY
23. GraphQL AI Working Group Recap: January 2026, accessed June 14, 2026, https://graphql.org/blog/2026-01-22-recap-jan-2026-ai-wg/
24. ai-wg/rfcs/semantic-introspection.md at main · graphql/ai-wg - GitHub, accessed June 14, 2026, https://github.com/graphql/ai-wg/blob/main/rfcs/semantic-introspection.md
25. TOON vs. JSON vs. YAML: Token Efficiency Breakdown for LLM - Medium, accessed June 14, 2026, https://medium.com/@ffkalapurackal/toon-vs-json-vs-yaml-token-efficiency-breakdown-for-llm-5d3e5dc9fb9c
26. YAML vs. JSON: Which Is More Efficient for Language Models? - Better Programming, accessed June 14, 2026, https://betterprogramming.pub/yaml-vs-json-which-is-more-efficient-for-language-models-5bc11dd0f6df
27. BAML vs POML vs YAML vs JSON for LLM Prompts | Augment Code, accessed June 14, 2026, https://www.augmentcode.com/learn/baml-vs-poml-vs-yaml-vs-json-for-llm-prompts

 © 2026 Vigensis

28. I benchmarked LEAN vs JSON vs YAML for LLM input. LEAN uses 47% fewer tokens with higher accuracy : r/Rag - Reddit, accessed June 14, 2026, https://www.reddit.com/r/Rag/comments/1sl50sq/i_benchmarked_lean_vs_json_vs_yaml_for_llm_input/
29. Best Structured Prompt Formats for LLMs, Ranked - MightyBot, accessed June 14, 2026, https://mightybot.ai/blog/best-structured-prompt-formats-for-llms/

 © 2026 Vigensis