A framework for efficient and secure LLM agency: a case for the GraphQL paradigm

Viktor Zhakhalov

Department of Artificial Intelligence Systems, Institute of Computer Science and Information Technologies, Lviv Polytechnic National University, 12 Stepana Bandery Str., Lviv, 79013, Ukraine

Abstract

LLM agents must translate natural language into concrete actions on external tools. Most systems use JSON-based function calling or, more riskily, let models emit imperative code. We propose a GraphQL-first alternative that reframes tool use as typed, declarative program synthesis against a schema. This yields three measurable advantages. First, efficiency: a token–economy analysis shows that a single GraphQL query replaces multiple RPC calls, reducing request tokens from 63 to 32 and total operational tokens from 245 to 175 in a representative user–orders task. Second, reliability: schema validation provides deterministic, structured error diagnostics that enable self-correcting interaction loops without bespoke prompt engineering. Third, security: the schema-bounded language forms a native sandbox that eliminates arbitrary code execution pathways and reduces prompt-injection impact to a bounded query surface governed by depth/complexity limits and authorization. Because GraphQL is standardized and widely understood by general coding LLMs, the approach is model-agnostic and interoperable. We argue that GraphQL constitutes a principled, testable alternative to function calling for agentic systems, combining lower cost, stronger safety, and improved cognitive robustness.

Keywords

GraphQL, LLM agents, declarative querying, function calling, token economy, validation, security

1. Introduction

1.1. The GraphQL paradigm for LLM-powered agents

The proliferation of large language model (LLM) powered agentic systems has introduced a new set of architectural challenges centered on the interface between the model's cognitive processes and the external world of tools and data [1]. The design of this interface is not a peripheral concern; it is a critical determinant of an agent's efficiency, security, and reliability. This analysis posits that GraphQL, a query language and server-side runtime for APIs, represents a fundamentally superior architectural choice for mediating these interactions compared to conventional REST-based approaches or direct code-execution frameworks [2]. Its core design principles – a strongly-typed schema, a single operational endpoint, and client-specified data fetching – directly address the primary challenges of cost, performance, and flexibility inherent in modern agentic systems.

1.2. Foundational principles and architectural advantages for agentic systems

GraphQL was conceived to solve the inefficiencies of traditional API designs, but its architecture is exceptionally well-suited to the unique demands of LLM agents [3]. First, GraphQL operates through a single, unified endpoint for all queries, mutations, and subscriptions [3]. For an LLM agent, which may need to orchestrate complex tasks involving multiple data sources, this model radically simplifies the interaction logic. The agent directs all requests to one predictable location, with the operation's nature defined entirely within the request payload, reducing the cognitive load of managing a sprawling API surface [4].

CS&SE@SW 2025: 8th Workshop for Young Scientists in Computer Science & Software Engineering, November 21, 2025, Kryvyi Rih, Ukraine
https://doi.org/10.55056/cssesw2025/paper07
viktor.v.zhakhalov@lpnu.ua (V. Zhakhalov)
0009-0002-3933-8562 (V. Zhakhalov)

Second, and most critically for an AI agent, every GraphQL API is defined by a strongly-typed schema [3]. This schema serves as an unambiguous contract, creating a self-documenting system that is not just for human developers but for the agent itself [5]. The agent can introspect this schema to understand precisely what actions it can perform and what data it can access, effectively giving it a definitive map of its capabilities and eliminating ambiguity [5]. This strong typing enables a crucial feature for agentic reliability: pre-execution validation. The GraphQL server can validate and reject any syntactically incorrect or schema-noncompliant query before it is ever executed, preventing a class of errors that would otherwise require complex handling logic [6].

This validation mechanism creates a powerful feedback system for the agent. When a query is invalid, the server returns a well-defined, structured error response that details exactly which fields or arguments were incorrect. Because this error format is standardized and predictable, an LLM can parse it, understand its mistake, and formulate a corrected query [7]. This enables the creation of robust, self-healing agentic loops where the agent can iteratively refine its requests based on structured feedback, learning to interact correctly with its environment without human intervention [5]. This structured, iterative correction is far more reliable than attempting to parse ambiguous, free-text error messages from traditional APIs and provides a foundation for building sophisticated autocorrection layers.

2. Related works: a comparative analysis of programming paradigms in the context of agentic AI

The interface language an LLM uses to act – imperative code, declarative query, or RPC – determines expressivity, reliability, and security. Beyond bespoke tool-calling formats, it is useful to place GraphQL alongside representative, actively maintained technologies: imperative (Python), declarative data query (SQL), RPC frameworks (gRPC/JSON-RPC), and declarative API query (GraphQL). We argue that a typed, declarative, non–general-purpose query language such as GraphQL offers a practical, bounded action surface for agents with strong validation and demand-control guidance [8, 9, 10].

2.1. Imperative code generation (Python)

Imperative generation gives a model maximum freedom over control flow and library usage, which can be attractive for complex orchestration or bespoke data transformation. Python exemplifies this flexibility: it is general-purpose, batteries-included, and supported by a vast ecosystem that an agent could, in principle, compose at runtime [11]. From a capability standpoint, this covers nearly any operation an agent might need.

However, this also maximizes risk. If an application executes model-emitted code, prompt-injection pathways can escalate to remote code execution (RCE) unless stringent isolation, syscall mediation, and policy constraints are applied [12, 13]. Even with containerization, the security burden remains high and errors are late-detected at execution time; by contrast, schema-validated query languages enable earlier (pre-execution) rejection of malformed or unauthorized actions.

2.2. Declarative data querying (SQL)

SQL remains the canonical declarative language for structured data access, with an actively maintained international standard (SQL:2023) and comprehensive production documentation (e.g., PostgreSQL 17) [14, 15]. For agents, SQL offers predictable semantics and powerful optimization, and it maps naturally to many enterprise data tasks.

The challenge is safe synthesis from natural language. Directly executing raw model-produced SQL risks injection and over-privilege; robust parameterization, least-privilege roles, and allow-listed operations are necessary [16]. Moreover, SQL alone does not provide an application-level capability model across heterogeneous tools; a schema-bounded API layer can centralize validation and authorization at the right abstraction level.

2.3. RPC frameworks (gRPC and JSON-RPC)

RPC exposes named procedures that an agent can invoke. gRPC provides a modern, typed IDL (Protocol Buffers), code-generated stubs, and streaming semantics, making it excellent for service-to-service communication at scale [17]. JSON-RPC 2.0 standardizes a light-weight, transport-agnostic message format that is easy to implement across languages [18].

In agent settings, however, method-per-action APIs often require multiple round trips to assemble nested data, increasing token and latency budgets relative to a single declarative query. Security and validation also live at each method boundary, which can fragment policy and complicate global demand control. A single, typed query surface reduces this sprawl and shifts validation earlier in the lifecycle.

2.4. Declarative API query (GraphQL)

GraphQL specifies a typed query language and validation/execution model, explicitly not a general-purpose programming language [8]. For agents, this means the "action space" is constrained to the schema: operations are declared, structured, and pre-validated before resolver execution. Introspection and predictable error messages further support iterative self-correction loops without bespoke error parsing.

From a security standpoint, GraphQL's risks are bounded and well-documented – depth and complexity abuse, batching, and authorization mistakes [9, 10]. These are systematically addressable via query depth/complexity limits, pagination, rate limiting, and field/operation-level authZ. The result is a narrow, auditable capability algebra that aligns with agent needs for precise data access and controlled mutations, while avoiding the unbounded hazards of arbitrary code execution.

2.5. Synthesis across paradigms for Agentic AI

Across paradigms, there is a clear trade-off between expressivity and safety. Imperative languages maximize what an agent could do but require heavy isolation to manage what it should be allowed to do. SQL offers strong declarative power for data, yet it operates at the database layer without an application-level capability model. RPC provides clear entry points but tends to fragment validation and authorization across many methods and round trips.

GraphQL's typed, declarative design centralizes capability description, validation, and demand control at a single endpoint. It is rich enough to express complex, nested interactions; bounded enough to pre-validate, monitor, and secure; and standardized enough for broad model interoperability. The comparison in table 1 consolidates paradigm, expressivity, common interaction shape (and token/latency implications), primary risks, and recommended mitigations [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18].

Viewed across table 1, GraphQL occupies a pragmatic middle ground. Compared with imperative code, it removes arbitrary execution pathways while preserving rich composition of nested operations with pre-execution validation [8]. Compared with SQL, it moves capability governance to the application/API layer and spans heterogeneous backends, while still allowing precise field selection that reduces over-fetching, tokens, and round-trip count [15, 19]. Relative to RPC, GraphQL's single endpoint and typed schema reduce surface sprawl and consolidate demand control and authorization policies [9, 10, 17, 18].

For Agentic AI, the implication is operational as much as architectural: a schema-bounded, declarative action language lowers the cost of safe autonomy. It streamlines tool understanding via introspection, enables deterministic error feedback for self-correction, and narrows the attack surface to a finite, auditable capability algebra. This combination is difficult to replicate with ad-hoc function calling or code execution without imposing heavier isolation burdens and higher token/latency overheads [12, 13, 19].

3. A superior token economy in agentic systems

In the context of LLM-powered applications, every token – whether in the input prompt or the model's generated output – carries a direct computational and financial cost. An agent's efficiency is therefore

Table 1
Paradigm-level comparison for LLM agent interaction.

Category	Representative	Paradigm	Strengths / interaction shape	Primary risks	Typical mitigations
Imperative code	Python [11]	Multi-paradigm (imperative / object-oriented / functional)	Maximal expressivity; arbitrary control flow and libraries; often one "big" action with many code tokens	Prompt-to-code → RCE if executed; late error detection [12, 13]	Strong sandboxing (containers/VM), syscall mediation, allow-listed APIs, policy engines
Declarative data query	SQL [14, 15]	Declarative relational query language	Mature, optimized data access; predictable semantics; may require multiple queries across services	Injection/over-privilege when synthesizing raw SQL [16]	Parameterization; least-privilege roles; allow-listed statements; row/column-level security
RPC frameworks	gRPC / JSON-RPC [17, 18]	IDL-based RPC (gRPC) / JSON message RPC	Typed stubs; streaming (gRPC); language-agnostic; method-per-action often means multiple round trips	Each method exposes imperative side effects; fragmented validation/authZ	Per-method validation; centralized authN/Z; request shaping; rate limits; idempotency
Declarative API query	GraphQL [8]	Typed query/mutation against a schema	Exact field selection; single round trip for nested data; pre-execution validation and structured errors	Bounded query abuse (depth/-complexity/-batching), auth mistakes [9, 10]	Depth/complexity limits; pagination; demand control; field/operation-level authZ; rate limiting

inextricably linked to its token economy. GraphQL's design provides a significant advantage in this domain, optimizing token usage at two critical stages: the definition of the agent's tools (the schema) and the execution of its actions (the query-response cycle).

3.1. Schema definition verbosity

Before an LLM agent can use a tool, it must be informed of the tool's existence, its purpose, and its parameters. This information is typically provided within the LLM's context window, often as part of a system prompt, using a structured format like JSON Schema for REST-based function calling or GraphQL's Schema Definition Language (SDL) for a GraphQL API. The verbosity of this schema definition directly impacts the baseline token cost of every interaction with the agent.

GraphQL's SDL is inherently more concise than JSON Schema [20, 4]. It employs a clean, human-readable syntax that eschews the boilerplate characteristic of JSON, such as excessive quotation marks, commas, and curly braces for structuring blocks [4]. Consider a simple schema for a Character type. In GraphQL SDL, this is expressed with clarity and brevity:

```
type Character {
  name: String!
  appearsIn: [Episode!]!
}
```

A corresponding definition in JSON Schema is substantially more verbose, requiring a nested object structure with explicit type, properties, required, and items keys for the same semantic information. This syntactic overhead, when multiplied across dozens or hundreds of tools available to a sophisticated

agent, results in a significant and persistent inflation of the input prompt's token count. A leaner schema definition not only reduces the cost of every LLM call but also frees up valuable space in the finite context window, allowing for more tools, richer examples, or longer conversational history.

3.2. Operational efficiency

The token savings afforded by GraphQL become even more pronounced during the operational lifecycle of a query and its response. The ability to specify the exact data fields required minimizes the token count of both the request sent to the API and, more importantly, the data payload returned to the LLM for processing [19].

This can be illustrated with a practical scenario: an agent tasked with retrieving a user's name, email, and the titles of their last three orders. The comparative token analysis demonstrates the difference in efficiency between a GraphQL implementation and a typical multi-endpoint REST implementation for this task.

Tool Schema/Definition – GraphQL (Token Count: 61)

```
type Query {
    user(id: ID!): User
}

type User {
    id: ID!
    name: String!
    email: String!
    address: String
    orders(last: Int): [Order!]
}

type Order {
    id: ID!
    title: String!
    date: String
    status: String
}
```

Tool Schema/Definition – JSON/REST (Token Count: 112)

```
[
    {
        "name": "getUser",
        "parameters": {
            "type": "object",
            "properties": {
                "userId": {
                    "type": "string
    "
                }
            }
        }
    },
    {
        "name": "getOrders",
        "parameters": {
            "type": "object",
            "properties": {
                "userId": {
                    "type": "string
    "
                },
                "limit": {
                    "type": "
    integer"
                }
            }
        }
    }
]
```

Query (Token Count: 32)

```
query {
    user(id: "123") {
        name
        email
        orders(last: 3) { title }
    }
}
```

Function Call (Token Count: 63)

```
[
    {
        "name": "getUser",
        "arguments": { "userId":
    "123" }
    },
    {
        "name": "getOrders",
        "arguments": {
            "userId": "123",
            "limit": 3
        }
    }
]
```

Response (Token Count: 48)

```
{
    "data": {
        "user": {
            "name": "John Doe",
            "email": "
    johndoe@example.com",
            "orders": [
                { "title": "Order
    #101" },
                { "title": "Order
    #102" },
                { "title": "Order
    #103" }
            ]
        }
    }
}
```

Function Results (Token Count: 22 + 82)

```
{ "name": "John Doe", "email": "
    johndoe@example.com" }

{
"orders": [
    { "title": "Order #101" },
    { "title": "Order #102" },
    { "title": "Order #103" }
]
}
```

Total Operational Tokens – GraphQL: 175; JSON/REST: 245.

As the comparison shows, the GraphQL query produced by the LLM is nearly half the size of the function-call request (32 vs. 63 tokens), while the resulting payloads are of similar order (48 vs. 82 tokens). The function-call variant requires two separate calls and returns over-fetched data, whereas the GraphQL variant completes the task with a single, precise query and returns only the requested fields [19].

This reduction in both request and response tokens lowers cost and latency and yields a cleaner, more focused context for subsequent reasoning. By excluding irrelevant fields, GraphQL reduces distraction and confusion for the model – a cognitive benefit that improves the reliability of the entire agentic system.

4. Democratizing tool generation for generalist LLMs

A key advantage of adopting GraphQL is its capacity to be understood and generated by a wide range of LLMs, particularly those with strong general coding capabilities but without specific fine-tuning for proprietary tool-calling formats. The syntax of a GraphQL query is highly structured, predictable, and

shares similarities with common programming constructs like JSON or object property access, making it a natural fit for models pre-trained on vast repositories of code.

Research has demonstrated that LLMs can achieve high accuracy in generating complex GraphQL queries from natural language prompts using only in-context learning [21]. This process typically involves providing the model with the relevant parts of the GraphQL schema and a few illustrative examples (few-shot prompting). The model can then generalize from this context to construct valid queries for new, unseen requests, obviating the need for costly and resource-intensive fine-tuning [22].

This stands in contrast to bespoke JSON-based tool-calling mechanisms, which often represent an arbitrary convention specific to a particular model or provider. Achieving reliable performance with such formats may require more elaborate prompt engineering or dedicated model training. By adopting a standardized, widely understood language like GraphQL, organizations can decouple their agentic architecture from any single LLM provider. This fosters a more flexible and model-agnostic ecosystem, where any sufficiently capable coding LLM can be harnessed to interact with the defined toolset. GraphQL thus serves as a universal adapter for tool use, leveraging the raw intelligence of generalist models and promoting greater interoperability across the AI landscape.

5. A comparative security analysis: declarative safety vs. imperative risk

The architectural choice of an agent's interaction language extends beyond efficiency and flexibility; it is a fundamental security decision with profound implications. The paradigm used to connect an LLM to external tools dictates the system's attack surface and its resilience to malicious manipulation. A critical analysis reveals a stark dichotomy: the bounded, declarative safety of GraphQL versus the unbounded, imperative risk inherent in frameworks that permit LLMs to generate and execute raw code.

5.1. The unbounded threat of arbitrary code execution

A prominent paradigm in agentic systems is the Code-First Approach, exemplified by libraries such as Hugging Face's smolagents [12]. This approach empowers an LLM to generate imperative code – typically Python – to perform actions and interact with its environment [12]. This design offers maximal expressivity, allowing the agent to perform complex logic, chain operations, and dynamically compose functions [12].

However, this flexibility comes at a severe security cost. By design, this architecture grants a probabilistic, non-deterministic text generator direct control over a privileged execution environment. The LLM is connected to critical organizational resources, including internal databases, networks, and third-party applications [23]. The inherent danger of this model is acknowledged by its proponents, who recommend executing the LLM-generated code within sandboxed environments like Docker or E2B to mitigate risk [12]. Yet, this reliance on sandboxing is itself an admission of the architecture's intrinsic vulnerability. It introduces significant operational complexity, performance overhead, and the ever-present risk of sandbox-escape exploits, shifting the security burden rather than eliminating the root cause of the threat [12].

5.2. Prompt injection as a vector for remote code execution

The primary attack vector against these code-executing agents is prompt injection. This class of attack involves crafting malicious user input that is designed to override the LLM's original instructions, tricking the model into performing unintended actions [23]. Because LLMs cannot reliably distinguish between developer-intended instructions and attacker-supplied instructions commingled within the prompt, they can be turned into confused deputies, faithfully executing the attacker's will [23]. In a system where the LLM's output is treated as executable code, a successful prompt injection attack can escalate directly to remote code execution (RCE) [13].

A typical sequence:

1. The vulnerable application uses a library like LangChain or LlamaIndex to generate Python from a user prompt and executes it with `eval`/`exec` [23].
2. The attacker submits a malicious prompt embedding instructions, e.g., `__import__('os').system('ls')` or `os.system('cat /etc/passwd')` [23, 24].
3. The LLM, as a confused deputy, generates the malicious code.
4. The backend executes that string, resulting in server compromise (RCE).

Public analyses document RCE via prompt injection patterns and related vulnerabilities in LLM-integrated apps [13]. The fundamental architectural flaw is granting the LLM's text output the privilege of execution, creating a direct, insecure pipeline from an untrusted source to high-privilege system functions.

5.3. GraphQL as a natively secure interaction layer

In stark contrast, GraphQL offers a natively secure, declarative interaction layer. It functions as a native sandbox, not through containerization, but through the inherent constraints of its language and schema. The entire set of possible actions is explicitly enumerated and bounded by the Query and Mutation types defined in the schema. The LLM's capabilities are restricted to this declared set. It is syntactically and semantically impossible for an LLM to generate a GraphQL query that executes arbitrary shell commands, because no such operation can be expressed in the language or defined in a sane server schema.

GraphQL APIs do have security concerns – e.g., DoS via deeply nested queries, batching, or improper introspection [10]. But these are bounded and manageable via rate limiting, query depth/complexity analysis, and robust authentication/authorization within resolvers [3, 10]. Securing a GraphQL-based agent is a bounded problem of access control and resource management; securing a code-executing agent is the unbounded problem of preventing arbitrary command execution from untrusted text.

6. Conclusion

This work introduces a scientific reframing of LLM tool use: from emitting ad-hoc JSON RPCs to synthesizing a single, typed, declarative program in GraphQL. Treating the agent's "act" step as query synthesis against a strongly typed schema yields pre-execution validation with deterministic error diagnostics, enabling reliable self-correction loops that are difficult to realize with heterogeneous function-calling formats. Because the action language is standardized and schema-bounded, the agent's behavior is constrained by an explicit capability algebra, reducing the attack surface from arbitrary code/RPC patterns to a finite, auditable set of operations while remaining model-agnostic and interoperable.

Empirically, the declarative formulation improves the token economy and execution efficiency: in a representative user–orders task the LLM's request shrinks from 63 to 32 tokens and total operational tokens from 245 to 175, while consolidating multiple calls into a single round trip. This combination of typed semantics, structured validation, and measurable cost reductions establishes GraphQL not just as an engineering convenience but as a principled, testable alternative to function calling for agentic systems.

Declaration on Generative AI

During the preparation of this work, the author used Google Gemini (including Gemini 2.5 Pro and Gemini Deep Research), Perplexity AI, and the OpenAI API (GPT-5) as follows:

- **Literature support:** Gemini and Perplexity AI were used to search for, organize, and compare related research and programming paradigms, and to surface relevant references for subsequent manual inspection.

- **Drafting support:** Gemini Deep Research was used to assist in outlining and drafting preliminary text fragments (e.g., candidate formulations, example structures, and alternative phrasings) for parts of the manuscript.
- **Editing and LaTeX preparation:** GPT-5 was used to assist with rephrasing, grammar and spelling checks, and with improving the structure and formatting of the LaTeX source (e.g., section organization, table layout, and minor wording edits).

All core research contributions, including the conception and design of the GraphQL-based agent framework, its implementation and deployment, the experimental analysis, and the scientific interpretations and conclusions, were developed by the author. After using these tools, the author reviewed, rewrote, and edited all AI-assisted content as needed and takes full responsibility for the content of this publication.

References

[1] M. Cekic, AI agent vs LLM: 5 differences and when to use each, 2026. URL: https://bito.ai/blog/ai-agent-vs-llm/.

[2] OpenReplay Team, MCP vs REST vs GraphQL: how llm-first apis are different, 2025. URL: https://blog.openreplay.com/mcp-rest-graphql-llm-first-apis/.

[3] S. Agarwal, GraphQL and Large Language Models (LLMs): A Powerful Combination for the Future of APIs and AI, Medium, 2025. URL: https://tinyurl.com/3hmf7mxh.

[4] T. Gopal, Why you should use GraphQL instead of a JSON DSL, 2018. URL: https://hasura.io/blog/why-not-use-a-json-dsl-instead-of-graphql-d29f20cc97d2.

[5] Z. Yang, GraphQL for LLM API Queries: Basics, 2025. URL: https://www.newline.co/@zaoyang/graphql-for-llm-api-queries-basics--157e85c0.

[6] The GraphQL Foundation, Validation, 2026. URL: https://graphql.org/learn/validation/.

[7] AgenticFlow AI: ChatGPT in the Flow of Work, Validators, 2025. URL: https://web.archive.org/web/20251019002953/https://docs.agenticflow.ai/visual-workflows-drag-and-drop-automation/actions-get-started/llm/validators.

[8] The GraphQL Foundation, GraphQL: Current Working Draft, 2026. URL: https://spec.graphql.org/draft/.

[9] The GraphQL Foundation, Security, 2026. URL: https://graphql.org/learn/security.

[10] N. Jones, S. Gerlach, A Developer's Guide to GraphQL Security Best Practices, 2026. URL: https://www.stackhawk.com/blog/graphql-security/.

[11] Python Software Foundation, General Python FAQ – What is Python?, 2026. URL: https://docs.python.org/3.15/faq/general.html#what-is-python.

[12] B. Williams, Autonomous AI Agents: A Hidden Risk in Insecure smolagents "CodeAgent" Usage, 2025. URL: https://www.nccgroup.com/research-blog/autonomous-ai-agents-a-hidden-risk-in-insecure-smolagents-codeagent-usage/.

[13] T. Liu, Z. Deng, G. Meng, Y. Li, K. Chen, Demystifying RCE Vulnerabilities in LLM-Integrated Apps, in: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 1716–1730. doi:10.1145/3658644.3690338.

[14] ISO/IEC, ISO/IEC 9075-1:2023: Information Technology – Database Languages SQL – Part 1: Framework (SQL/Framework), 2023. URL: https://www.iso.org/standard/76583.html.

[15] The PostgreSQL Global Development Group, PostgreSQL: Documentation: 17: Part II. The SQL Language, 2026. URL: https://www.postgresql.org/docs/17/sql.html.

[16] kingthorin, zbraiterman, SQL Injection, 2026. URL: https://owasp.org/www-community/attacks/SQL_Injection.

[17] gRPC Authors, Introduction to gRPC, 2024. URL: https://grpc.io/docs/what-is-grpc/introduction/.

[18] JSON-RPC Working Group, JSON-RPC 2.0 Specification, 2013. URL: https://www.jsonrpc.org/specification.

[19] M. Hawkins, Every Token Counts: Building Efficient AI Agents with GraphQL and Apollo MCP Server, 2025. URL: https://www.apollographql.com/blog/building-efficient-ai-agents-with-graphql-and-apollo-mcp-server.

[20] DerNalia, Why Should I Choose GraphQL over JSON:API?, 2017. URL: https://www.reddit.com/r/reactjs/comments/74mquk/why_should_i_choose_graphql_over_jsonapi/.

[21] B. Ganesan, S. Ghosh, N. Gupta, M. Kesarwani, S. Mehta, R. Sindhgatta, LLM-powered GraphQL Generator for Data Retrieval, in: K. Larson (Ed.), Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24, International Joint Conferences on Artificial Intelligence Organization, 2024, pp. 8657–8660. doi:10.24963/ijcai.2024/1002, Demo Track.

[22] M. Kesarwani, S. Ghosh, N. Gupta, S. Chakraborty, R. Sindhgatta, S. Mehta, C. Eberhardt, D. Debrunner, GraphQL Query Generation: A Large Training and Benchmarking Dataset, in: F. Dernoncourt, D. Preoţiuc-Pietro, A. Shimorina (Eds.), Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track, Association for Computational Linguistics, Miami, Florida, US, 2024, pp. 1595–1607. doi:10.18653/v1/2024.emnlp-industry.117.

[23] N. Mousseri, Oligo ADR in Action: LLM Prompt Injection, 2024. URL: https://www.oligo.security/blog/oligo-adr-in-action-llm-prompt-injection.

[24] A. Kang, Fake (Hallucinated) Remote Code Execution (RCEs) in LLM Applications, Cyber Defense Magazine, 2025. URL: https://www.cyberdefensemagazine.com/fake-hallucinated-remote-code-execution-rces-in-llm-applications/.