

Architecting a Unified Tool-Calling Layer for Heterogeneous LLM Ecosystems

Executive Summary

The capacity for Large Language Models (LLMs) to interact with external systems via "tool calling" represents a fundamental shift, transforming them from passive text generators into active, autonomous agents. This capability is the cornerstone of modern AI applications, enabling them to access real-time data, execute actions, and overcome inherent knowledge limitations. However, the proliferation of LLM providers has led to a fragmented and non-standardized landscape of tool-calling APIs. Major providers, including OpenAI, Google, and Anthropic, have implemented powerful but mutually incompatible interfaces, creating significant engineering overhead and fostering vendor lock-in. This report presents a comprehensive architectural blueprint for a **Unified Communication Layer** designed to abstract these differences and provide a consistent, provider-agnostic interface for building robust, multi-provider LLM applications.

The proposed architecture is centered on a canonical, internal representation of tools and tool calls, managed by three core components: the **Adapter**, the **Parser**, and the **Executor**. The Adapter translates a standardized tool definition into the specific format required by any target LLM. The Parser normalizes the LLM's response, detecting and translating tool-call invocations into a consistent internal format. The Executor then manages the execution of the corresponding application logic. This design effectively decouples the application's core agentic logic from the idiosyncrasies of any single provider's API.

This report provides a detailed analysis of the strategic trade-offs involved in this architecture. It strongly recommends a syntax-rewriting approach within the Parser to ensure true application independence. Furthermore, it offers a dual strategy for model support: leveraging the reliability of native tool-calling features for capable models, while employing a sophisticated, multi-tiered simulation strategy for models that lack this built-in functionality. This simulation approach ranges from advanced prompt engineering to more robust methods like API-native JSON modes and constrained decoding. By adopting the principles and patterns outlined herein, organizations can build a scalable, resilient, and future-proof foundation for developing advanced agentic systems in a complex and evolving LLM ecosystem.

1.0 Introduction: The Fragmentation of Tool-Calling

Standards

1.1 The Strategic Importance of Tool Use in Modern LLM Applications

The evolution of Large Language Models has reached a critical inflection point. Beyond their initial capabilities in natural language generation, their true potential is unlocked when they can interact directly with external systems, databases, and APIs.¹ This interaction is facilitated by a mechanism known as "tool calling" or "function calling." This capability is not merely an incremental feature; it is the essential bridge that transforms LLMs from information retrievers into functional agents capable of performing tasks in the digital and physical worlds.³

Tool use is the primary method for grounding LLMs, connecting their vast but static internal knowledge to dynamic, real-time information sources. It allows an LLM-powered application to answer a query like "What is the current weather in Tokyo?" not by recalling potentially outdated training data, but by invoking a live weather API.⁴ This overcomes the inherent "knowledge cutoff" problem that limits the utility of standalone models.⁵

The applications of this paradigm are vast and transformative. They range from building sophisticated conversational agents that can book appointments or manage calendars², to automating complex business processes by interacting with internal enterprise software.⁷ Furthermore, tool calling is fundamental to creating advanced data extraction and analysis pipelines, where an LLM can interpret an unstructured request, formulate a structured database query, execute it, and synthesize the results into a human-readable summary.⁴ In essence, tool use is the enabling technology for the entire field of agentic AI.

1.2 The Core Challenge: Divergent APIs and Syntaxes Across Providers

As the strategic importance of tool use became apparent, all major LLM providers—including OpenAI, Google, Anthropic, Cohere, and Mistral—rushed to integrate native tool-calling capabilities into their models.⁵ While this widespread adoption validates the importance of the feature, it has created a significant challenge for application developers: the APIs are conceptually similar but syntactically incompatible.⁹

This divergence is not merely a matter of different endpoint URLs or authentication methods. It extends to the core data structures used to define tools and represent their invocation. For example, OpenAI's API expects tool definitions in a `tools` parameter with a specific JSON Schema structure, while Anthropic's API uses a similar `tools` parameter but requires the schema to be nested under an `input_schema` key.⁹ The format of the model's response is equally varied. OpenAI returns a `tool_calls` object containing a JSON *string* for arguments, whereas Google Gemini returns a `functionCall` object with a native JSON *object* for its

arguments, and Anthropic signals a tool invocation via a `stop_reason` of `tool_use` and places the call details in a distinct content block.⁹

This fragmentation presents a formidable obstacle for organizations aiming to build provider-agnostic applications. It forces developers to write brittle, provider-specific logic, effectively leading to vendor lock-in. If an application is built around OpenAI's specific tool-calling syntax, migrating to Anthropic's Claude or Google's Gemini requires a substantial rewrite of the core agentic loop.¹³ This technical friction is not accidental; it represents a strategic competitive battleground where providers create ecosystem moats through API design. By making their interfaces unique, they increase the switching costs for developers, thereby capturing and retaining users within their platform. A unified communication layer is therefore not just a technical convenience but a strategic necessity for maintaining architectural flexibility and avoiding long-term dependency on a single provider.

1.3 Defining the Unified Communication Layer: Goals and Principles

To counteract this fragmentation, this report proposes the architecture of a Unified Communication Layer. This layer is a form of middleware that sits between the core application logic and the diverse ecosystem of LLM providers, presenting a single, consistent interface for all tool-related interactions.

The primary goals of this unified layer are:

- **Define Once, Use Anywhere:** To allow developers to define the logic and schema for a tool a single time, in a canonical, provider-agnostic format. The layer is then responsible for translating this definition into the specific format required by any target LLM.
- **Standardized Invocation:** To provide a uniform method for detecting when an LLM has requested a tool call and to parse that invocation into a consistent internal data structure, regardless of the source model's native syntax.
- **Seamless Model Switching:** To enable developers to swap the underlying LLM provider (e.g., from OpenAI's GPT-4o to Anthropic's Claude 3.5 Sonnet) with minimal to no changes in the application's agentic code, thereby facilitating performance benchmarking and cost optimization.¹³
- **Graceful Degradation:** To architecturally support a hybrid environment of models, including those with full native tool-calling capabilities and those without, which require the feature to be simulated through advanced prompting techniques.

To achieve these goals, the design of the layer should adhere to several key architectural principles:

- **Modularity:** The components responsible for handling each specific provider should be isolated and self-contained, allowing them to be updated or replaced without affecting the rest of the system.
- **Extensibility:** The architecture must make it simple to add support for new LLM providers as they emerge, requiring only the implementation of a new provider-specific module that adheres to a common internal interface.

- **Resilience:** The layer must be robust against common failure modes, such as malformed JSON or unexpected API responses from the LLM, and provide clear, structured error handling.
- **Transparency:** The layer should offer comprehensive logging and observability hooks to allow developers to inspect the translated API calls and normalized responses, aiding in debugging and performance tuning.

2.0 Foundational Concepts: The Anatomy of a Tool Call

Before designing the unified layer, it is essential to establish a clear conceptual model of the tool-calling process. This process can be universally broken down into a three-stage lifecycle, regardless of the specific LLM provider.

2.1 The Three Pillars: Definition, Invocation, and Response

The end-to-end lifecycle of a tool call consists of the following distinct stages:

1. **Definition (Client-Side):** This is the process of describing a tool's capabilities to the LLM. The developer provides a schema that typically includes the tool's name, a natural language description of its purpose, and a structured definition of its input parameters.¹⁰ This stage is a critical and powerful extension of prompt engineering; a well-written description and a clear parameter schema significantly improve the model's ability to decide when and how to use the tool correctly.¹ The tool schemas are passed to the model as part of the API request.
2. **Invocation (Model-Side):** Based on the user's prompt and the provided tool definitions, the LLM uses its reasoning capabilities to determine if invoking one or more tools would be beneficial for fulfilling the user's request. If it decides to proceed, it does not execute any code itself. Instead, it generates a structured data object—typically in JSON format—that represents a request to call a specific tool with a specific set of arguments.³ This structured output is the "tool call" and is returned as part of the API response.
3. **Execution & Response (Client-Side):** The application code receives the API response containing the tool call invocation. It is the application's responsibility to parse this structured data, identify the requested tool and its arguments, and execute the corresponding function or code. After the tool has been executed, its output (the "tool result") is formatted and sent back to the LLM in a subsequent API call. The LLM then uses this result to synthesize a final, user-facing natural language response.¹²

2.2 A Comparative Analysis of Native Tool-Calling APIs

The primary challenge for a unified layer lies in the syntactic and structural differences across provider APIs at the Definition and Invocation stages. A detailed technical comparison reveals these key points of divergence.

2.2.1 OpenAI: The tools and tool_calls Paradigm

- **Definition:** Tools are passed in a tools array in the API request. Each tool object must have type: "function" and a function object containing a name, description, and a parameters object that adheres to the JSON Schema specification.⁸ More recent API versions introduce a strict: true flag within the function definition, which guarantees that the model's output will conform to the provided JSON Schema, enhancing reliability.¹⁶
- **Invocation Output:** When a tool is called, the response message object from the assistant contains a tool_calls array. Each element in this array is an object with a unique id, a type of "function", and a function object. This nested object contains the name of the function to be called and an arguments field, which is a JSON-formatted *string* that must be parsed by the client.⁸

2.2.2 Google Gemini: OpenAPI Schema and functionCall

- **Definition:** Gemini also uses a tools parameter, which contains an array of functionDeclarations. The schema for each function is defined using a subset of the OpenAPI 3.0 schema, specifying a name, description, and parameters.¹² The structure is conceptually very similar to OpenAI's but is explicitly framed within the OpenAPI standard.
- **Invocation Output:** A model response that includes a tool call will contain a part with a functionCall object. This object directly contains the name of the function and an args field. Crucially, args is a structured JSON *object*, not a string, which eliminates the need for the client to perform an extra JSON parsing step and reduces a potential point of failure.¹²

2.2.3 Anthropic Claude: input_schema and tool_use Blocks

- **Definition:** Claude's API accepts a tools parameter where each tool object contains a name, description, and an input_schema. The input_schema serves the same purpose as OpenAI's parameters and Gemini's parameters, defining the function's arguments using JSON Schema.¹¹
- **Invocation Output:** Claude signals a tool call differently. The API response will have a stop_reason field with the value "tool_use". The content array of the response will then

contain one or more blocks of type: "tool_use". Each of these blocks represents a distinct tool call and contains a unique id, the name of the tool, and an input object containing the arguments as a structured JSON object.¹¹

2.3 The Role of Schemas: JSON Schema as a Lingua Franca

Despite the syntactic differences in the surrounding API structure, a critical point of convergence exists: all major providers have adopted **JSON Schema** as the standard for defining the parameters of a tool.¹⁰ This shared foundation is the key that makes a unified abstraction layer feasible. JSON Schema provides a rich, standardized vocabulary for describing the structure of the data a tool expects, including data types, required fields, descriptions, and constraints like enums.

This convergence allows the unified layer to adopt JSON Schema as the core of its "canonical tool definition." The task of the provider-specific adapters then becomes a relatively straightforward translation of the surrounding metadata (e.g., mapping the canonical parameters key to Claude's input_schema key) rather than a complex transformation of the schema itself.

Furthermore, the quality of the JSON Schema definition is paramount to the performance of the tool-calling system. Providing clear, descriptive names for parameters, detailed descriptions of their purpose, and constraints such as enums where applicable, is a form of "schema-driven prompt engineering." It gives the model strong signals that guide its reasoning process, significantly increasing the likelihood that it will invoke the correct tool with valid arguments.³

The following table provides a direct, at-a-glance comparison of these API differences, highlighting the specific translation and parsing logic that a unified layer must implement.

Feature	OpenAI	Google Gemini	Anthropic Claude
Tool List Parameter	tools	tools	tools
Tool Definition Object	{ "type": "function", "function": {...} }	{ "functionDeclarations": [...] }	{ "name":..., "description":..., "input_schema":... }
Parameter Schema Key	parameters	parameters	input_schema
Response Indicator	message.tool_calls is not null	Response part contains functionCall	stop_reason is "tool_use"
Tool Call Location	message.tool_calls (array)	content.parts (array contains functionCall object)	content (array contains tool_use blocks)
Tool Call ID	message.tool_calls[n].id	Not explicitly provided in the same way	content[n].id
Tool Name Location	message.tool_calls[n].function.name	functionCall.name	content[n].name

Argument Location	message.tool_calls[n].function.arguments	functionCall.args	content[n].input
Argument Format	JSON String	JSON Object	JSON Object

3.0 Architectural Blueprint for a Unified Communication Layer

Based on the foundational concepts, a robust and extensible architecture for the unified layer can be designed. This architecture is fundamentally an implementation of the **Adapter design pattern**, a classic software engineering solution for making incompatible interfaces work together. Recognizing this allows for the application of established best practices for building flexible and maintainable systems. The core principle is to define a stable internal interface for tool interactions, while treating the provider-specific logic as interchangeable plugins (adapters).

3.1 Core Components: The Adapter, Parser, and Executor

A high-level architectural diagram of the unified layer reveals four primary components:

- **Schema Registry:** This is a central, in-memory or persistent repository that stores the canonical definitions of all available tools. It acts as the single source of truth for tool schemas within the application.
- **Adapter:** For each supported LLM provider, there is a corresponding Adapter module. Its sole responsibility is to handle outgoing requests. It takes a list of canonical tool definitions from the Schema Registry and translates them into the specific format and structure required by the target LLM's API. This includes mapping key names (e.g., parameters to input_schema) and arranging the data into the correct request body structure. This process is encapsulated in a method one might call to_provider_spec.
- **Parser:** The Parser is the counterpart to the Adapter, responsible for handling incoming responses from the LLM. Each provider has a dedicated Parser module. It inspects the raw API response, determines if any tool calls have been made, extracts all relevant information (tool name, arguments, call ID), and normalizes this data into a provider-agnostic, canonical ToolCall object. This is handled by a method like from_provider_spec.
- **Executor:** This component is provider-agnostic and operates on the standardized output from the Parser. It receives a list of canonical ToolCall objects, looks up the corresponding application function (e.g., via a name-to-function mapping), executes the function with the provided arguments, and formats the return value into a canonical ToolResult object, ready to be sent back to the LLM in the next turn of the

conversation.¹

3.2 The Canonical Tool Definition: A Provider-Agnostic Schema

The cornerstone of the unified layer is the canonical tool definition. This is a standardized internal data structure used to represent a tool, independent of any provider. Leveraging the convergence around JSON Schema, the proposed canonical schema is straightforward:

JSON

```
{
  "name": "string",
  "description": "string",
  "parameters": {
    "type": "object",
    "properties": {... },
    "required": [... ]
  }
}
```

In practice, developers should not have to write this JSON by hand. High-level abstractions can be used to define tools in a more ergonomic way. Frameworks like LangChain and LlamaIndex provide excellent examples of this approach. Tools can be defined as:

- **Decorated Python Functions:** The function name, docstring, and type hints are automatically introspected to generate the canonical schema.⁵
- **Pydantic Models:** A Pydantic class can define the schema, with field names, types, and descriptions serving as the source for the parameters object.¹⁴
- **TypedDicts or Similar Structures:** Other typed data structures in various languages can serve the same purpose.¹⁴

The unified layer is responsible for this introspection, converting the developer-friendly definition into the canonical JSON format stored in the Schema Registry.

3.3 The Registration Flow: Translating Canonical Definitions

The process of making tools available to an LLM for a specific call follows a clear, adapter-driven flow:

1. A developer defines a set of tools using a high-level abstraction (e.g., a Python function decorated with `@tool`).
2. The unified layer's introspection mechanism parses these definitions and populates the

Schema Registry with their canonical JSON representations.

3. When the application makes a call to the LLM (e.g., `llm.invoke(prompt, tools=['get_weather'])`), it specifies the target provider.
4. The layer selects the appropriate Adapter for that provider.
5. The Adapter retrieves the canonical schemas for the requested tools from the Schema Registry.
6. It then performs the translation: it might rename the `parameters` key to `input_schema` for Claude, or wrap the definition inside a `{ "type": "function", "function": ... }` object for OpenAI.
7. Finally, the Adapter injects this provider-specific tool list into the final API request that is sent to the LLM.

This entire flow is exemplified by LangChain's `.bind_tools()` method, which elegantly handles this translation behind the scenes, allowing the developer to write the same code regardless of the underlying model.¹

3.4 The Invocation Flow: Detecting, Parsing, and Normalizing

The reverse process for handling the LLM's response is equally critical:

1. The raw HTTP response is received from the LLM provider's API.
2. The layer invokes the Parser corresponding to that provider.
3. The Parser inspects the response for provider-specific tool call indicators. For Anthropic, it checks if `stop_reason == "tool_use"`. For OpenAI, it checks if the `message.tool_calls` array is present and non-empty.
4. The Parser then navigates the provider-specific JSON structure to extract the tool name, arguments, and a unique call ID for each invocation.
5. It performs any necessary data type conversions, such as parsing the JSON string in OpenAI's `arguments` field into a dictionary.
6. Finally, it normalizes this information into a list of canonical `ToolCall` objects. A standardized internal representation could look like this: `[{"id": "call_abc123", "name": "get_weather", "args": {"location": "London", "unit": "celsius"}}]`.

This list of normalized `ToolCall` objects is the final output of the unified layer for a given turn, which is then passed to the Executor. LangChain's `AIMessage.tool_calls` attribute is a perfect real-world implementation of this normalized output, providing a consistent list of `ToolCall` objects regardless of the model that generated them.⁹

3.5 Error Handling and Resilience: Managing Malformed Outputs

A production-grade system must anticipate and handle failures gracefully. LLMs, particularly those without strong native fine-tuning for tool use, can occasionally produce malformed outputs, such as arguments that are not valid JSON or are missing required fields.⁵ The

unified layer should incorporate a robust error-handling strategy.

- **Schema Validation:** After the Parser extracts the arguments, the Executor should always validate them against the tool's canonical JSON Schema before attempting to execute the function.
- **Structured Error Representation:** If parsing or validation fails, the layer should not crash. Instead, it should populate a structured error object. LangChain's `InvalidToolCall` object, which can contain the tool name, the malformed arguments string, an ID, and an error message, is an excellent pattern to follow.¹⁴ This allows the application to decide how to handle the failure.
- **Automated Repair Loops:** A sophisticated strategy for handling validation failures is to implement a repair loop. The application can catch the structured error, format a new prompt that includes the original request, the model's malformed output, and the validation error message, and send this back to the LLM with an instruction to "fix the malformed tool call." This can often resolve the issue without human intervention.²⁷

4.0 Handling Models with Native Tool-Calling Capabilities

For models that have first-class, built-in support for tool calling, the implementation of the unified layer focuses on the precise translation logic within the Adapter and Parser components.

4.1 Implementing Provider-Specific Adapters

The core logic of the unified layer resides in the provider-specific modules. The following outlines the conceptual implementation for each major provider:

- **OpenAI Adapter & Parser:**
 - **Adapter (to_provider_spec):** Takes a list of canonical tool schemas. For each schema, it constructs an object { "type": "function", "function": schema }. This list is then assigned to the tools key in the API request payload.
 - **Parser (from_provider_spec):** Checks the response for a message.tool_calls array. If present, it iterates through the array. For each element, it extracts the id, function.name, and function.arguments. It performs a `json.loads()` operation on the arguments string to convert it into a dictionary. It then constructs a canonical `ToolCall` object.
- **Anthropic Claude Adapter & Parser:**
 - **Adapter (to_provider_spec):** Takes a list of canonical tool schemas. For each schema, it creates a new object by copying the name and description and renaming the parameters key to `input_schema`. This list of translated objects is

assigned to the tools key in the API request.

- **Parser (from_provider_spec):** Checks if the response's stop_reason is "tool_use". If so, it filters the content array for blocks where type == "tool_use". For each such block, it extracts the id, name, and the input object (which is already a dictionary) to construct the canonical ToolCall object.
- **Google Gemini Adapter & Parser:**
 - **Adapter (to_provider_spec):** Takes a list of canonical tool schemas. It wraps this list in an object with a functionDeclarations key: { "functionDeclarations": schemas }. This object is then placed inside the main tools array in the request.
 - **Parser (from_provider_spec):** Iterates through the content.parts array in the response. It looks for a part that contains a functionCall key. If found, it extracts the name and the args object to construct the canonical ToolCall object.

4.2 Real-Time Syntax Rewriting vs. Direct Passthrough: A Trade-off Analysis

A critical architectural decision is whether the unified layer should always enforce its canonical format or simply pass through the provider's native output.

- **Syntax Rewriting (Recommended):** In this model, the Parser always transforms the provider's native tool call syntax into the layer's internal canonical format. The application's core logic *only* ever interacts with this canonical format.
 - **Pros:** This achieves true decoupling. The application code is completely insulated from provider-specific details, making it highly portable and maintainable. Conditional logic like if provider == 'openai':... is eliminated from the main business logic.
 - **Cons:** It introduces a marginal amount of processing overhead for the transformation. It may also abstract away certain provider-specific metadata that is not part of the canonical format, though this is rare for core tool-calling features.
- **Direct Passthrough:** In this alternative, the layer could expose the raw provider output directly to the application.
 - **Pros:** There is zero transformation overhead, and the application has access to every detail of the provider's response.
 - **Cons:** This approach fundamentally defeats the primary purpose of a unified layer. It pushes the complexity of handling different formats up into the application logic, leading to tangled, hard-to-maintain code and destroying provider-agnosticism.

For any system aiming for scalability, maintainability, and flexibility, **syntax rewriting is the unequivocally superior architectural choice**. The minor overhead is a small price to pay for the immense benefit of a clean, abstracted, and portable agentic core.

4.3 Managing Advanced Features: Parallel and Sequential Tool Calls

LLM capabilities are not uniform. A key differentiator is the ability to perform parallel tool calling—that is, to request multiple, independent tool invocations within a single turn.²⁸ For example, in response to "What is the weather in Tokyo and the current stock price of Google?", a model with parallel capabilities could generate two tool calls simultaneously. The unified layer must handle this heterogeneity gracefully.

- The Parser for each provider should be designed to handle both single and multiple tool calls transparently. Whether the provider returns one tool call or an array of them, the Parser's output should *always* be a list of canonical ToolCall objects. If there is only one call, it will be a list with a single element.
- This design choice simplifies the downstream Executor and application logic immensely. The code can simply iterate over the list of tool calls it receives, without needing to know whether the underlying model supports parallel execution or not.¹¹ This pushes the complexity of handling provider differences into the Parser, where it belongs, and keeps the application logic clean and simple. However, developers must be aware of potential issues that can arise with parallel calls, such as the model returning duplicate tool_call_ids or attempting to call tools it does not have access to in a specific context.²⁹

4.4 Leveraging Frameworks: A Deep Dive into LangChain's Standardized Interface

Building a unified layer from scratch is a significant engineering effort. Fortunately, mature frameworks like LangChain have already implemented this architecture, providing a battle-tested foundation upon which to build.³⁰

LangChain's standardized tool-calling interface is a premier example of the architectural principles described in this report.

- **ChatModel.bind_tools()** serves as the **Adapter**. This method accepts tools in various high-level formats (decorated functions, Pydantic models) and handles the translation to the provider-specific schema required by the model it's bound to.⁹ This allows the same tool definition to be used with an OpenAI, Anthropic, or Google model interchangeably.⁹
- **AIMessage.tool_calls** is the output of the **Parser**. This attribute, attached to the model's response message, provides a standardized list of ToolCall objects. It abstracts away whether the original response came from OpenAI's tool_calls array or Anthropic's tool_use content blocks, presenting a single, consistent interface to the developer.⁹
- **create_tool_calling_agent()** is a higher-level constructor that demonstrates the power of this abstraction. Because it is built on top of the standardized bind_tools and tool_calls interfaces, it can create a functional agent that works with *any* compliant

model, replacing older, provider-specific agent constructors.⁹ For most projects, the most pragmatic approach is not to reinvent this infrastructure but to either build directly on top of a framework like LangChain or to adopt its proven design patterns when constructing a custom layer.

5.0 Simulating Tool Calls for Models Without Native Support

A truly comprehensive unified layer must also support models that lack first-class, fine-tuned tool-calling capabilities, such as many open-source or older proprietary models. For these models, the feature must be "simulated" by guiding the model to produce a structured output that mimics a native tool call. This shifts the responsibility of the Adapter from modifying API parameters to performing sophisticated prompt engineering.

5.1 The Prompt Engineering Approach: System Prompts and Few-Shot Examples

The foundational technique for simulating tool calls is to use the prompt to instruct the model on the desired behavior.³¹ The Adapter for a non-native model will construct a detailed system prompt that includes several key elements:

- **Role-Playing Instruction:** The prompt begins by assigning the model a role, such as "You are a helpful assistant that can use external tools to answer questions".³²
- **Tool Definitions:** The full schemas of all available tools are included directly in the prompt, typically formatted as a JSON or XML block for clarity. The description of each tool and its parameters is crucial for the model to understand its function.³¹
- **Formatting Instructions:** The prompt must give explicit, unambiguous instructions on the exact format the model should use when it decides to call a tool. For example: "When you need to use a tool, you must respond with a single JSON object containing two keys: 'tool_name' with the name of the tool to call, and 'arguments' with an object containing the parameters."³⁴
- **Few-Shot Examples:** To dramatically improve reliability, the prompt can include one or more examples of a user query followed by the correctly formatted tool-call JSON.³⁴

This in-context learning helps the model generalize the required output format. While this approach can work, its reliability can be inconsistent, as the model is still generating free-form text and may fail to adhere perfectly to the specified format.²⁷

5.2 Structured Output Generation: Moving Beyond Basic Prompting

To address the unreliability of pure prompt engineering, more robust methods can be employed to force the model to generate structured output.

5.2.1 API-Native JSON Modes and Structured Output Features

Many modern LLM APIs, even for models not explicitly fine-tuned for tool calling, now offer a "JSON mode" or a more general "Structured Output" feature.³⁷ When this mode is enabled via an API parameter (e.g., `response_format: { "type": "json_object" }`), the provider guarantees that the model's entire output will be a syntactically valid JSON string.³⁷

The unified layer's Adapter should detect if the target model's API supports this feature and enable it. This is far more reliable than simply asking for JSON in the prompt.³⁶ The prompt still needs to define the tools and instruct the model to generate a JSON object with the desired `tool_name` and `arguments` keys, but the API itself enforces the JSON syntax. The layer's Parser is still required to validate that the content of the JSON matches the expected schema.

5.2.2 Constrained Decoding and Logit Biasing with External Libraries

For open-source models hosted locally (e.g., via vLLM, Ollama, or llama.cpp), the generation process itself can be controlled directly. Libraries like outlines and Microsoft's Guidance framework leverage a technique called constrained decoding.²⁷

This method works by intervening at each token generation step. Before the model selects the next token, the library uses a Finite State Machine (FSM) or a regular expression derived from the target JSON Schema to determine the set of *valid* next tokens. It then modifies the model's output probabilities (logits) to force it to pick only from that valid set.⁴⁰ This approach *guarantees* that the final output will conform to the specified schema, making it the most reliable method for simulating tool calls with non-native models. The unified layer's Adapter for such models would integrate with one of these libraries to apply the constraints during generation.

5.3 The Two-Stage Invocation Pattern: Decision-Making then Parameter Generation

When an agent has access to a large number of complex tools, including all of their full schemas in the prompt can consume a significant portion of the context window and potentially confuse the model. A more advanced pattern, the two-stage invocation, can improve both reliability and efficiency.³⁶

1. **Stage 1 (Decision):** The first LLM call is made with a prompt that includes only the *names and descriptions* of the available tools, not their full parameter schemas. The

model is instructed to perform a decision-making task: either answer directly or choose a single tool to use. The expected output is a simple, constrained JSON object, such as { "reasoning": "...", "answer": "...", "use_tool": "tool_name" | null }.

- 2. **Stage 2 (Parameter Generation):** If the model decides to use a tool, the application logic initiates a second LLM call. This time, the prompt includes the original user query and the full JSON Schema for *only the selected tool*. The model's task is now much simpler: generate the arguments for that specific tool. This stage should be combined with a constrained decoding method to guarantee a valid output.

While this pattern introduces additional latency and cost from the second LLM call, it offers significant benefits: the initial prompt is much smaller, reducing token usage, and each LLM call has a much simpler, more focused task, which dramatically increases the accuracy and reliability of the outcome.³⁶

5.4 Validation and Repair: Ensuring Schema Compliance for Simulated Calls

Regardless of the simulation method used, the validation and repair mechanisms discussed in Section 3.5 become even more critical. Simulated calls are inherently more prone to error than native, fine-tuned capabilities. A robust validation step against the canonical JSON Schema is non-negotiable, and an automated repair loop that feeds errors back to the model is a highly effective strategy for building a resilient system.

The following table summarizes the trade-offs between these simulation strategies, providing a decision-making framework for architects.

Strategy	Description	Reliability	Implementation Complexity	Best For...
Basic Prompting	Instructing the model via system prompts to generate a specific JSON format.	Low	Low	Quick prototypes, non-critical applications.
Few-Shot Prompting	Providing examples of correct input/output pairs in the prompt.	Medium	Low	Improving the reliability of basic prompting with minimal effort.
API-Native JSON Mode	Using a provider's built-in feature to guarantee valid JSON syntax.	High (Syntax), Medium (Schema)	Low	Models that support it but lack full tool-calling fine-tuning.
Constrained	Using external	Very High	High	Production

Decoding	libraries (e.g., outlines) to force schema compliance during token generation.			systems using open-source/local models.
Two-Stage Invocation	A two-step process: first decide which tool to use, then generate its parameters.	Very High	High	Complex agents with a large number of available tools.

6.0 Advanced Orchestration and Agentic Design

A unified tool-calling layer is not an end in itself; it is the foundational infrastructure upon which sophisticated, multi-step agentic systems are built. Handling a single tool call is a building block. The true power emerges when an application can orchestrate sequences of these calls to solve complex problems that require planning, reasoning, and state management.

6.1 Beyond Single Invocations: Building Stateful, Multi-Step Workflows

Most non-trivial tasks require more than one tool call. For example, booking a flight might involve first searching for available flights (tool 1), then checking the user's calendar for conflicts (tool 2), and finally booking the selected flight (tool 3). This requires an "agent loop" where the LLM repeatedly assesses the situation, calls a tool, receives the result, and reasons about the next step until the overall goal is accomplished.³⁵

However, simple while loops can be brittle. They can get stuck in repetitive cycles, lose track of the overall goal, or fail to handle unexpected tool outputs gracefully.¹³ This motivates the need for more structured and controllable orchestration frameworks.

6.2 Introduction to Graph-Based Execution Models (e.g., LangGraph)

A more robust way to model complex agentic workflows is as a state graph.³⁰ Frameworks like LangGraph allow developers to define the agent's logic as a directed graph where:

- **Nodes** represent actions or computations. A node could be "call the LLM," "execute tools," or any custom Python function.

- **Edges** represent the conditional transitions between nodes. After a node completes, logic on the edges determines which node to move to next.

For example, after an "call the LLM" node, a conditional edge could check the LLM's output. If it contains tool calls, the graph transitions to a "execute tools" node. If it contains a final answer, the graph transitions to an "end" state. This provides explicit, fine-grained control over the agent's flow, enabling complex behaviors like loops, branching, and human-in-the-loop approval steps.²⁴

The unified communication layer is a critical enabler for this paradigm. By providing a standardized, reliable output (the canonical ToolCall object), it allows for the creation of generic, reusable nodes like a ToolNode that can execute any tool call produced by any model.²⁴ This abstraction allows developers to focus on the high-level logic of the agent's workflow (the graph structure) rather than the low-level, provider-specific implementation details of tool calling. This shift from an imperative while loop to a declarative graph definition represents a significant maturation in agentic architecture, and it is only made possible by the underlying abstraction of the unified layer.

6.3 Managing Context and Tool Results in Multi-Turn Conversations

In a multi-step workflow, managing the conversation history is paramount. After a tool is executed by the Executor, its result must be communicated back to the LLM so it can proceed with its reasoning. This is done by appending a new message to the conversation history. However, just as with tool *calls*, the required format for tool *results* differs between providers.

- **OpenAI** expects a message with the role set to tool, containing the content of the tool's output and the tool_call_id from the original request.
- **Anthropic** expects a user message that contains a content block of type: "tool_result", which includes the tool_use_id and the content.

The unified layer's **Adapter** must therefore also be responsible for translating a canonical ToolResult object (e.g., { "call_id": "...", "content": "..." }) into the correct provider-specific message format before sending it in the next API request. This ensures that the full round-trip of tool call and result is abstracted away from the application logic.

6.4 Designing "Deep Agents": Combining Planning, Sub-Agents, and Memory

The most capable agents, often referred to as "deep agents," can tackle complex, long-horizon tasks like conducting in-depth research or writing software. Their capabilities are built upon the foundation of tool calling, but they combine it with more advanced architectural patterns.³⁵

- **Planning:** Instead of reacting turn-by-turn, a deep agent might first use a tool call to generate a step-by-step plan (e.g., calling a create_todo_list tool). It then executes this

plan, providing a more structured and robust approach to problem-solving.

- **Sub-Agents:** Complex tasks can be broken down and delegated. A main "orchestrator" agent can use tool calls to invoke specialized sub-agents. For example, a research agent might call a "data-scraping" sub-agent and a "data-analysis" sub-agent. Each sub-agent is, itself, a tool that the orchestrator can call.
- **Memory/File System:** For tasks that generate a large amount of intermediate context, relying solely on the LLM's context window is inefficient and limiting. Deep agents use tools to interact with an external memory, such as a virtual file system or a database. They can write notes, save intermediate results, and read back information as needed, effectively extending their working memory far beyond the limits of the context window.³⁵

The unified tool-calling layer is the essential fabric that enables all of these advanced patterns. It provides the standardized API that planners, orchestrators, and memory systems use to interact with the LLM's core reasoning engine.

7.0 Synthesis and Strategic Recommendations

The analysis of the tool-calling landscape and the architectural blueprint for a unified communication layer leads to a set of clear, actionable recommendations for architects and engineers building multi-provider LLM applications.

7.1 Recommendation: Adopt a Canonical Schema with Provider-Specific Adapters

The core architectural recommendation is to build the system around a provider-agnostic, canonical representation for tool definitions and tool calls. This internal standard should be based on JSON Schema for parameter definition. The interaction with each LLM provider should then be encapsulated within dedicated, modular Adapters and Parsers. This approach, rooted in the classic Adapter design pattern, provides the most robust, maintainable, and future-proof foundation. It isolates provider-specific complexity, simplifies application logic, and ensures that the system can be easily extended to support new models as the ecosystem evolves.

7.2 Recommendation: Prioritize Models with Native Tool Calling and Structured Output Guarantees

For production systems where reliability and predictability are paramount, the selection of the underlying LLM is critical. Architects should strongly prefer models that have been explicitly

fine-tuned for tool use and offer first-class, native support for the feature. Furthermore, the availability of features that *guarantee* schema compliance, such as OpenAI's strict: true option for structured outputs, should be a significant factor in model selection.¹⁶ These features dramatically reduce the likelihood of runtime errors from malformed outputs and lessen the burden on the application's validation and repair logic. Simulated tool-calling approaches should be reserved for prototyping, research, or applications involving specialized open-source models where constrained decoding can be directly implemented.

7.3 Recommendation: Employ a Hybrid Strategy for Syntax Enforcement

The question of whether to rewrite syntax in real-time or enforce it via prompting does not have a single answer; it depends on the capability of the target model. The optimal strategy is a hybrid one:

- **For Models with Native Support:** Rely entirely on the unified layer's **Parser** to perform real-time syntax rewriting. The application should receive a canonical ToolCall object, and the Adapter should handle the formatting of the ToolResult message. This is the most reliable and efficient method.
- **For Models Without Native Support:** Use **prompt engineering** (including system prompts, formatting instructions, and few-shot examples) as the primary mechanism to *guide* the model toward the correct output format. However, this guidance should not be trusted implicitly. It must be coupled with a mechanism to *enforce* structure. Where available, use API-native JSON modes. For local models, use **constrained decoding**. In all cases, a strict validation and repair loop is essential to handle the inherent unreliability of this simulation approach.

7.4 Future Outlook: The Convergence of APIs and the Evolution of Agent Frameworks

The current fragmentation of tool-calling APIs is characteristic of a rapidly maturing but still nascent technology sector. Looking forward, a gradual convergence towards a more standardized API format is likely, probably centering on the OpenAPI specification, as the ecosystem stabilizes and interoperability becomes a stronger market demand.

As this low-level standardization occurs, the focus of innovation and product differentiation will continue to shift "up the stack." The key challenges will move from the mechanics of a single tool call to the complex orchestration of multi-step, multi-agent workflows.

Consequently, declarative, graph-based agent frameworks like LangGraph and the agent orchestration features in Semantic Kernel will become increasingly critical tools for building the next generation of AI systems.³⁰ The unified communication layer, therefore, is not merely a tactical solution to today's interoperability problems. It is a necessary strategic investment,

providing the stable, abstracted foundation required to build the sophisticated and powerful agentic architectures of tomorrow.

Works cited

1. Tool calling | 🦜 LangChain, accessed October 9, 2025, https://python.langchain.com/docs/concepts/tool_calling/
2. Tool Calling with Llama: Enhancing AI Capabilities, accessed October 9, 2025, <https://www.llama.com/resources/cookbook/toolcalling-with-llama/>
3. Function calling using LLMs - Martin Fowler, accessed October 9, 2025, <https://martinfowler.com/articles/function-call-LLM.html>
4. Function Calling with LLMs - Prompt Engineering Guide, accessed October 9, 2025, https://www.promptingguide.ai/applications/function_calling
5. How to do tool/function calling | 🦜 LangChain, accessed October 9, 2025, https://python.langchain.com/docs/how_to/function_calling/
6. Function Calling in Large Language Models (LLMs) | by Mehmet Ozkaya - Medium, accessed October 9, 2025, <https://mehmetozkaya.medium.com/function-calling-in-large-language-models-llms-8e7712c0e60f>
7. Claude Developer Platform, accessed October 9, 2025, <https://www.claude.com/platform/api>
8. OpenAI Function Calling Tutorial: Generate Structured Output | DataCamp, accessed October 9, 2025, <https://www.datacamp.com/tutorial/open-ai-function-calling-tutorial>
9. Tool Calling with LangChain - LangChain Blog, accessed October 9, 2025, <https://blog.langchain.com/tool-calling-with-langchain/>
10. Guide to Tool Calling in LLMs - Analytics Vidhya, accessed October 9, 2025, <https://www.analyticsvidhya.com/blog/2024/08/tool-calling-in-llms/>
11. Tool use with Claude - Anthropic, accessed October 9, 2025, <https://docs.anthropic.com/en/docs/build-with-claude/tool-use>
12. Function calling with the Gemini API | Google AI for Developers, accessed October 9, 2025, <https://ai.google.dev/gemini-api/docs/function-calling>
13. Why use Langchain, when OpenAI has multi step sequential tool calling and reasoning?, accessed October 9, 2025, https://www.reddit.com/r/LangChain/comments/1mmjp77/why_use_langchain_when_openai_has_multi_step/
14. How to use chat models to call tools | 🦜 LangChain, accessed October 9, 2025, https://python.langchain.com/docs/how_to/tool_calling/
15. How to use function calling with Azure OpenAI in Azure AI Foundry Models - Microsoft Learn, accessed October 9, 2025, <https://learn.microsoft.com/en-us/azure/ai-foundry/openai/how-to/function-calling>
16. Introducing Structured Outputs in the API - OpenAI, accessed October 9, 2025, <https://openai.com/index/introducing-structured-outputs-in-the-api/>
17. Partially structured output? Free text output, but force correct tool call JSON -

- API, accessed October 9, 2025,
<https://community.openai.com/t/partially-structured-output-free-text-output-butt-force-correct-tool-call-json/955147>
18. Introduction to function calling | Generative AI on Vertex AI - Google Cloud, accessed October 9, 2025,
<https://cloud.google.com/vertex-ai/generative-ai/docs/multimodal/function-calling>
 19. Generate content with the Gemini API in Vertex AI - Google Cloud, accessed October 9, 2025,
<https://cloud.google.com/vertex-ai/generative-ai/docs/model-reference/inference>
 20. Use function calling with Anthropic to enhance the capabilities of Claude | Generative AI on Vertex AI | Google Cloud, accessed October 9, 2025,
<https://cloud.google.com/vertex-ai/generative-ai/docs/samples/generativeai-on-vertex-ai-claude-3-tool-use>
 21. Claude 3.5: Function Calling and Tool Use - Composio, accessed October 9, 2025,
<https://composio.dev/blog/claude-function-calling-tools>
 22. Tool use with Claude, accessed October 9, 2025,
<https://docs.claude.com/en/docs/build-with-claude/tool-use/overview>
 23. Function Calling - Hugging Face, accessed October 9, 2025,
<https://huggingface.co/docs/hugs/guides/function-calling>
 24. Call tools - GitHub Pages, accessed October 9, 2025,
<https://langchain-ai.github.io/langgraph/how-tos/tool-calling/>
 25. Standardizing Tool Calling with Chat Models #20343 - GitHub, accessed October 9, 2025, <https://github.com/langchain-ai/langchain/discussions/20343>
 26. How to use chat models to call tools - LangChain.js, accessed October 9, 2025,
https://js.langchain.com/docs/how_to/tool_calling/
 27. Every Way To Get Structured Output From LLMs - BAML, accessed October 9, 2025, <https://boundaryml.com/blog/structured-output-from-llms>
 28. How to disable parallel tool calling | 🦜 LangChain, accessed October 9, 2025,
https://python.langchain.com/docs/how_to/tool_calling_parallel/
 29. Looking for ideas: How to handle parallel tool calls in LangGraph? : r/LangChain - Reddit, accessed October 9, 2025,
https://www.reddit.com/r/LangChain/comments/1dmtcyn/looking_for_ideas_how_to_handle_parallel_tool/
 30. Architecture - LangChain, accessed October 9, 2025,
<https://python.langchain.com/docs/concepts/architecture/>
 31. Achieving Tool Calling Functionality in LLMs Using Only Prompt Engineering Without Fine-Tuning Citation: Authors. Title. Pages.... DOI:0000000/11111. - arXiv, accessed October 9, 2025, <https://arxiv.org/html/2407.04997v1>
 32. Prompting VS function calling... what do you use? : r/OpenAI - Reddit, accessed October 9, 2025,
https://www.reddit.com/r/OpenAI/comments/1f8irms/prompting_vs_function_calling_what_do_you_use/
 33. Anthropic Claude Function Calling.ipynb - GitHub Gist, accessed October 9, 2025,

- <https://gist.github.com/markomanninen/7c6e1e96faf0ec4382249e2131a9f1d0>
34. How to implement function calling using only prompt? : r/LocalLLaMA - Reddit, accessed October 9, 2025, https://www.reddit.com/r/LocalLLaMA/comments/1anaatm/how_to_implement_function_calling_using_only/
 35. Deep Agents - LangChain Blog, accessed October 9, 2025, <https://blog.langchain.com/deep-agents/>
 36. Structured Output as a Full Replacement for Function Calling | by Vitaly Sem | Medium, accessed October 9, 2025, <https://medium.com/@virtualik/structured-output-as-a-full-replacement-for-function-calling-430bf98be686>
 37. Function Calling in the OpenAI API, accessed October 9, 2025, <https://help.openai.com/en/articles/8555517-function-calling-in-the-openai-api>
 38. Structured output | Gemini API - Google AI for Developers, accessed October 9, 2025, <https://ai.google.dev/gemini-api/docs/structured-output>
 39. The guide to structured outputs and function calling with LLMs - Agenta, accessed October 9, 2025, <https://agenta.ai/blog/the-guide-to-structured-outputs-and-function-calling-with-llms>
 40. Taming LLMs: How to Get Structured Output Every Time (Even for Big Responses), accessed October 9, 2025, <https://dev.to/shrsv/taming-llms-how-to-get-structured-output-every-time-even-for-big-responses-445c>
 41. Semantic Kernel Agent Architecture | Microsoft Learn, accessed October 9, 2025, <https://learn.microsoft.com/en-us/semantic-kernel/frameworks/agent/agent-architecture>