# Agentic API

**A Task-Centric Framework for Scalable Agent Integrations**

By Chris Hood

May 16, 2025

| Version | Author | Date | Changes |
|---|---|---|---|
| v 0.1 | Chris Hood | November 30, 2024 | Introductory concept |
| v 0.2 | Chris Hood | December 20, 2025 | Adjusted scope of framework |
| v 0.3 | Chris Hood | January 31, 2025 | Aligned with agent-based processes |
| v 0.4 | Chris Hood | February 28, 2025 | Comparison for MCP / A2A |
| v 0.5 | Chris Hood | April 4, 2025 | Draft completed |
| v 1.0 | Chris Hood | May 16, 2025 | Finalized version for sharing. |

# Executive Summary

As AI agents become embedded in enterprise systems, the interface between agents and services has come under scrutiny. While some propose agent-to-agent protocols like MCP or A2A to support coordination, these approaches address the wrong problem, introducing complexity without solving interface deficiencies (Wooldridge & Jennings, 1995). The core challenge is not a lack of inter-agent standards, but the absence of APIs designed for task execution and semantic clarity.

Existing APIs, built around CRUD operations, are optimized for data access, not for representing actionable intent (Fielding & Taylor, 2002). This forces agents to infer meaning from endpoints not designed for Artificial Intelligence (AI) and AI Agents use.

We propose AgenticAPI, a task-oriented interface specification that enables agents to discover, understand, and invoke system capabilities without protocol mediation. AgenticAPI supports Agent Experience (AX) by providing intuitive, action-oriented APIs that ensure seamless task execution for AI agents, akin to user and developer experiences. Built on existing API infrastructure, AgenticAPI introduces a standardized action model supporting intent expression, contextual execution, and composable workflows.

At its core is the ACTION taxonomy, comprising six task categories: Acquire, Compute, Transact, Integrate, Orchestrate, and Notify. These serve as semantically meaningful alternatives to CRUD, enabling contextually relevant agent interactions (Horvitz, 1999). The model supports verbs like search, summarize, recommend, book, and notify, detailed in Section 4 and Appendix A.

AgenticAPI aligns with principles including:

- **Contextual Alignment**: APIs represent actions within context, including availability.
- **Semantic Discoverability**: Verbs convey intent in machine-readable formats.
- **Execution Clarity**: Endpoints define preconditions and side effects.
- **Compatibility and Extensibility**: Extends OpenAPI conventions.
- **Intent Weighting**: Includes variables like priority or confidence.
- **Adaptive Output**: Supports multiple response formats, JSON, JS, Language blob.

AgenticAPI simplifies integration, enhances interoperability, and supports automation, particularly in finance, healthcare, and logistics. It maintains backward compatibility while establishing a foundation for agent-native systems. The ACTION model's alignment with human task language reduces translation overhead, enabling precise execution.



This paper contends that AI-system integration depends on APIs exposing intent and context, not agent-to-agent protocols. AgenticAPI and ACTION provide a pathway to effective, resilient, agent-driven architectures.

A proof of concept is in development to validate AgenticAPI's efficacy, testing reduced integration complexity and enhanced task execution. This PoC focuses on finance and healthcare, simulating standard tasks like booking to demonstrate agent usability and system scalability:

- **Basic Task**: Tests a standard task like `BOOK /meeting` to demonstrate task-oriented execution.
- **Agent Connection**: Connects an agent to AgenticAPI, invoking actions via `DISCOVER /actions`.
- **Output Validation**: Validates conversational and JSON outputs (e.g., meeting confirmation) for accuracy.
- **Error Reduction**: Measures error rates using `TEST /action` to ensure reliability.
- **Speed and Scale**: Evaluates execution speed and scalability under high-frequency requests.

 

# 1. Integration Has Changed, Our APIs Haven't

The history of systems integration is marked by successive efforts to reduce manual coordination and increase system interoperability. Early enterprise systems relied on tightly coupled modules, point-to-point custom integrations, or human-mediated workflows for data exchange and task execution. As businesses digitized and distributed architectures became more prevalent, integration strategies evolved in response to the growing need for automation, modularity, and scale.

The introduction of Service-Oriented Architecture (SOA) formalized the notion of encapsulated services that could be reused across systems. However, SOA's implementation was often hindered by its reliance on heavyweight standards (e.g., SOAP, WSDL) and centralized governance models that limited flexibility. The emergence of RESTful APIs represented a pragmatic departure from these earlier paradigms, emphasizing simplicity, statelessness, and uniform interfaces. REST, when coupled with HTTP and the CRUD (Create, Read, Update,

Delete) model, became the dominant pattern for web and system integration throughout the 2010s (Wikipedia, 2025).

REST APIs significantly lowered the barrier for integrating across systems, enabling developers to expose services and resources in a language-agnostic, platform-independent manner. Tools like Swagger (now OpenAPI), combined with JSON serialization, provided human-readable, machine-consumable specifications that became the backbone of modern software ecosystems.

Despite these advancements, REST and CRUD-based APIs were designed primarily for human developers, not intelligent systems. The structure and semantics of typical APIs assume the presence of a developer or tightly coupled application logic that understands the endpoint, interprets its parameters, and manually orchestrates workflows across calls (The New Stack, 2025). Even as tools have emerged to automate parts of this process (e.g., SDK generation, workflow engines), the design of the APIs themselves has remained fundamentally static and data-centric.

## The Rise of Agentic Interaction

The current wave of AI-based automation, particularly with the proliferation of AI agents, represents a significant departure from previous integration models. Unlike traditional applications that consume APIs in rigid, predefined ways, AI agents are expected to operate more flexibly. They simulate user behavior, reason across workflows, and interact with services based on goals, not just procedural logic. In many cases, they do so without explicit hardcoding, instead inferring actions from context, language, or prior training (Nordic APIs, 2025).

Agent Experience (AX) demands APIs that enable seamless task execution, such as booking meetings or summarizing reports, with machine-readable intent. AgenticAPI addresses AX by



delivering task-focused endpoints that agents can discover and invoke precisely, overcoming the semantic limitations of traditional interfaces.

Critically, these agents are task-oriented, not resource-oriented. Whereas CRUD-based APIs expose operations such as `GET /users` or `POST /orders`, agents typically require interfaces that align with goals such as “book a meeting,” “summarize this report,” or “notify the team lead.” These operations often span multiple API calls, require contextual awareness, and rely on higher-level semantics to be executed successfully (Microsoft Learn, 2024).

As a result, traditional API interfaces designed for granular resource manipulation do not expose sufficient intent-level semantics for agents to plan, execute, and complete complex tasks. The agent must interpret vague or inconsistent endpoint names, understand undocumented side effects, and chain together atomic operations with uncertain outcomes. This lack of expressiveness becomes a bottleneck in agentic automation and invites brittle, non-generalizable implementations.

## The Stagnation of Interface Semantics

While the execution environment and computational capabilities of AI agents have advanced rapidly, API interfaces themselves have largely stagnated in terms of semantic clarity. Most REST APIs continue to reflect internal data structures rather than external user intent. For example, endpoints like `GET /items`, `POST /purchase`, or `PUT /status` require a degree of contextual knowledge that must be manually encoded or inferred from documentation. The onus is placed on the consuming system to understand what an endpoint does, how it relates to a broader task, and under what conditions it should be used (The New Stack, 2025).

This disconnect creates a significant integration mismatch. Agents are expected to act on behalf of users or systems, yet the interfaces available to them lack the affordances necessary for informed action. The agent, therefore, must simulate developer-like behavior by reading documentation, inferring workflows, and recovering from ambiguous failures, rather than acting as an operational executor with clear, discoverable options.

## Toward a Language of Intent

To support agent-based automation, APIs must evolve beyond CRUD and expose task-oriented capabilities in a consistent and machine-interpretable format (Hood, 2024; C.D., 2022). This requires a shift in API design philosophy: from exposing data endpoints to exposing actions. In this model, the API does not simply offer access to records; it declares the operations that can be meaningfully performed within a given context (DZone, 2015; Better Programming, 2023).

A language of intent, structured around verbs rather than nouns, enables agents to align interface capabilities with high-level goals. For example, an agent attempting to perform a travel booking should not need to infer that `POST /flights` means “book a flight,” or that a `PATCH`

call to a calendar entry can be used to reschedule an appointment. Instead, interfaces should declare operations such as `BOOK /flight`, `SCHEDULE /meeting`, or `CANCEL /reservation`, allowing the agent to identify executable tasks without semantic translation.

This approach does not discard existing web standards or protocol conventions. Rather, it augments them by enriching the semantic layer exposed to consumers, particularly AI agents. Such a shift facilitates not only greater agent logic but also more robust developer tooling, better documentation practices, and improved system maintainability through contracts of intent.

The shift from human-coded automation to agent-driven task execution requires a corresponding shift in API design. While CRUD-based REST APIs have served well in data-centric architectures, they are insufficient for the demands of modern AI agents tasked with simulating real-world operations.

## The Constraint of Rigid Outputs

Traditional APIs typically deliver fixed response formats, such as JSON payloads, designed for developer parsing and UI rendering (Fielding & Taylor, 2002). However, AI agents, which operate across diverse tasks and contexts, require adaptive outputs to support dynamic workflows (Horvitz, 1999; Wooldridge & Jennings, 1995). An agent might need structured JSON for data analysis, a natural language summary for communication, or executable code for downstream automation. The static nature of JSON responses limits agents' ability to process outputs adaptively, forcing reliance on brittle parsing or post-processing logic.

In contrast, agentic interfaces must support flexible, context-sensitive outputs, including structured JSON, text blobs, or formats like JavaScript, PDFs, or natural language strings (Berners-Lee et al., 2001; Microsoft, 2024). Semantic web research emphasizes machine-interpretable, adaptive data representations such as JSON-LD (Lanthaler & Gütl, 2013). For example, an agent calling `SCHEDULE /meeting?output_type=natural_lang` might receive "*Your meeting with Jamie is confirmed for Thursday at 2 PM*," while `output_type=json` returns structured data for calendar integration. This multimodal capability aligns with mixed-initiative principles, where systems adapt outputs to user or agent intent (Horvitz, 1999).

Industry frameworks like Microsoft's Semantic Kernel already demonstrate this, enabling agents to process language, code, or data based on task needs. While standards like OpenAPI support extensible formats, they require further evolution to fully meet agentic requirements (OpenAPI Initiative, 2021). Without this flexibility, APIs risk becoming automation bottlenecks, forcing agents to work around rigid outputs (Gupta, 2025). To scale agent-native systems, outputs must be treated as dynamic artifacts of intent, tailored to execution context and downstream logic.

# 2. The Problem with Protocol-Led Thinking

As AI agents increasingly serve as intermediaries between users and digital systems, interest in communication protocols facilitating agent coordination has surged. Notable proposals include the Model Context Protocol (MCP), Agent Communication Protocol (ACP), and Agent-to-Agent (A2A) frameworks, which aim to standardize mechanisms such as discovery, message passing, and task delegation, reminiscent of early web service protocols for machine-to-machine interactions. While these protocols address interoperability in distributed systems, their development often stems from a flawed assumption: practical AI agent functionality requires peer-to-peer collaboration. Modern AI agents function as context-sensitive pattern matchers driven by inference, not negotiation (Wooldridge, 2020).

Anthropic's introduction of MCP frames the protocol as a universal standard to connect AI systems with data sources, to improve model responsiveness by eliminating integration silos. According to their announcement, "open technologies like the Model Context Protocol are the bridges that connect AI to real-world applications," designed to "replace today's fragmented integrations with a more sustainable architecture" (Anthropic, 2024). The MCP model defines client-server roles, where AI applications can retrieve context or interact with enterprise data through long-lived server connections, offering an abstraction layer over traditional system APIs.

The enthusiasm for agent-to-agent protocols reflects not an architectural necessity but a deficiency in interface design. For instance, MCP introduces complexities, such as long-lived state management, lack of robust authentication, and poor compatibility with stateless infrastructures like REST APIs or serverless functions, without addressing the core bottleneck: agents' inability to access semantically rich APIs for discovering and executing meaningful actions (Masood, 2025).

MCP's approach, which grants agents raw access to databases and file systems, assumes intelligence emerges from unrestricted data access. However, this bypasses critical safeguards like rate limiting, audit logging, and access control, which are standard in modern API ecosystems.

These shortcomings necessitate architectural workarounds that obscure responsibility, increase latency, and complicate recovery, particularly in asynchronous or multi-step workflows. What is often termed "agent collaboration" is, in practice, distributed control flow better suited to task-oriented interfaces than complex protocol stacks (Sun, 2025).

Instead of introducing new communication layers between agents, a more scalable solution is to evolve the API layer. By exposing clearly defined actions, embedding semantic metadata, and supporting contextual execution, APIs can become self-describing and machine-operable interfaces for agent-driven execution.



## Protocols as a Response to Interface Deficiency

Agentic protocols like MCP and ACP are designed to address challenges such as:

- How can one agent discover tools known to another?
- How can agents share user or task context?
- How can agents delegate or negotiate task ownership?
- How can agents invoke tools with unknown schemas?

While theoretically valid, these challenges are not unique to AI agents and do not necessitate novel protocol development. Instead, they highlight the absence of machine-interpretable, intent-expressive APIs that articulate system capabilities and their conditions and expected outcomes (Yang et al., 2025). The rise of agent-level protocols is thus a symptom of underpowered interface design rather than an architectural imperative. With robust APIs, agents could discover capabilities and execute actions without negotiation or delegation.

## The Myth of Agent Collaboration

The discourse surrounding agent protocols draws heavily from multi-agent systems (MAS) theory, where agents are modeled as autonomous entities with beliefs, desires, and reasoning capabilities. This perspective assumes agents coordinate to achieve distributed objectives. However, modern AI agents lack such autonomy. They do not formulate persistent plans, evaluate trade-offs, or engage in deliberative negotiation. Instead, their behavior resembles instructional pattern completion, responding to inputs, inferring next steps, and invoking tools (Hong et al., 2024).

What appears as agent collaboration is typically a sequence of tool calls within a single agent or workflow runner. These sequences do not require peer-to-peer messaging but relatively straightforward, callable interfaces that describe actions in terms of task intent, not raw data access (Sun, 2025). Over-reliance on MAS-inspired protocols risks misaligning system design with the practical capabilities of current AI agents.

## Structural Overhead and System Fragility

Inter-agent protocols introduce architectural complexity in several ways:

- **Discovery Overhead**: Agents must maintain registries of peers, service endpoints, and protocol capabilities, increasing latency and failure points.
- **Context Serialization**: Effective communication requires serializing and transmitting context—goals, task states, and execution history—in mutually understood formats, necessitating complex schema and vocabulary standardization.
- **Coordination Logic**: Protocols imply task negotiation, ownership transfer, or consensus mechanisms, which must be hard-coded, resulting in brittle interdependencies.

- **Debugging Complexity**: Tracing failures across distributed agent chains, reliant on ephemeral messages or stochastic inference, creates significant observability challenges.

These factors elevate agent integration's cognitive and operational burden, offering minimal performance or robustness gains compared to simpler, direct API invocation models (Liddle, 2025). Protocol-led architectures abstract the interface layer rather than enhance it.

## Clarifying the Role of Agents

Treating agents as peer systems rather than automation interfaces has driven architectural choices prioritizing theoretical completeness over practical utility. A grounded view positions agents as interfaces for intent interpretation and task execution, translating natural language or instructions into actionable system calls (Hong et al., 2024). Their role is not to communicate with each other via new protocols but to leverage clear APIs for action execution.

If APIs are designed with semantic metadata, contextual affordances, and standardized execution formats, agents' tasks become manageable. They can discover available operations, select based on relevance or constraints, and invoke them without protocol handshakes (Masood, 2025). The bottleneck shifts from inter-agent abstraction to interface clarity.

## Why Protocols Proliferate When Interfaces Fail

Protocols like MCP and ACP emerge to address coordination challenges that arise only when APIs are semantically opaque or underpowered. They are compensatory mechanisms, not foundational requirements, built on the premise that intent must be brokered through additional messaging layers. This paper advocates an alternative: evolving API design to expose actions, not just data, with machine-readable intent formalization. Such interfaces could eliminate the need for agent-to-agent communication in most cases, providing a scalable foundation for intelligent task execution.

# 3. A New Paradigm for API Design

For over two decades, API design has been dominated by the CRUD model, Create, Read, Update, Delete, implemented over HTTP as `POST`, `GET`, `PUT/PATCH`, and `DELETE`, respectively. These operations provide a simple, consistent interface for manipulating data resources, aligning with object persistence models and facilitating intuitive resource mapping in RESTful architectures (Fielding & Taylor, 2002). CRUD's conceptual simplicity enables developers to reason about system interactions through predictable, idempotent actions, supporting tooling standardization, automated SDK generation, and alignment with web architecture principles like statelessness and uniform interfaces.

Despite its strengths, CRUD's effectiveness wanes when APIs must support task execution rather than resource manipulation. As systems evolve to accommodate AI agents, automation frameworks, and context-aware orchestration, CRUD reveals limitations that constrain expressiveness, increase cognitive load, and misalign interfaces with user intent (Mouat, 2024). These shortcomings hinder the scalability and adaptability of agent-driven systems.

## CRUD's Operational Limitations

From a functional perspective, CRUD is insufficient for modeling the diversity and granularity of real-world operations that intelligent agents must perform. Consider the following scenarios:

- An agent is instructed to "recommend three relevant articles based on a user's reading history."
- A financial automation system must "analyze spending behavior and trigger alerts when anomalies are detected."
- A scheduling assistant is expected to "book the earliest available meeting slot with all required participants."

These tasks involve multi-step processes, conditional logic, or derived reasoning that cannot be naturally expressed through CRUD operations. Forcing them into CRUD leads to overloaded endpoints, ambiguous semantics, and ad hoc conventions. For example, a `POST /alerts` might variably "send," "schedule," or "trigger" an alert, lacking semantic transparency for consuming agents. Moreover, CRUD prioritizes data-centricity over capability expression, exposing what data is stored rather than what actions can be performed, forcing agents to reverse-engineer workflows by navigating resource structures.

## Toward an Intent-Centric API Model

To support agent-driven automation, APIs must be restructured around tasks, not tables. The unit of integration must shift from "resource" to "action." This demands a departure from CRUD's noun-based framing and the adoption of a verb-oriented interface language, to a model that exposes intent, supports contextual variation, and reflects real-world operations in machine-readable terms.

To address this need, we introduce the **ACTION** framework, a six-category taxonomy of API operations designed to support task-level interaction:

1. **Acquire:** Retrieve information with contextual filters or purpose-driven queries. Includes operations such as `search`, `scan`, `monitor`, or `extract`.

2. **Compute:** Transform, analyze, or summarize data using embedded or declarative logic. Encompasses verbs like `summarize`, `calculate`, `validate`, or `rank`.

3. **Transact:** Execute operations that alter system state or confirm commitments. Includes `purchase`, `book`, `cancel`, `register`, or `approve`.

4. **Integrate:** Combine or synchronize information across services or domains. Verbs include `merge`, `sync`, `map`, or `link`.

5. **Orchestrate:** Manage workflows involving sequencing, conditions, or parallel execution. Covers `schedule`, `chain`, `batch`, or `retry`.

6. **Notify:** Communicate updates, results, or alerts to systems or users. Includes `notify`, `alert`, `broadcast`, or `escalate`.

Unlike CRUD’s symmetrical operations, the ACTION framework embraces asymmetry, recognizing that not all systems support all verbs or expose all resources directly (Allamaraju, 2023). This taxonomy prioritizes executable affordances, enabling agents to interact with systems based on task intent.

## Designing APIs That Act, Not Just Serve

By adopting ACTION as the basis for API design, developers can construct interfaces that are more easily interpreted, invoked, and composed by agents. The emphasis shifts from exposing data models to declaring what the system can do, in what context, and with what constraints.

This shift introduces several design advantages:

- **Semantic Precision**: An endpoint such as `RECOMMEND /articles` conveys task intent directly, reducing the need for implicit assumptions or out-of-band documentation.
- **Contextual Variation**: The same verb can be specialized through parameterization (e.g., `SUMMARIZE /document?id=123&mode=bullets`) or scoped behaviorally to reflect environmental factors.
- **Chaining and Composition**: Actions can be composed predictably by agents when verbs expose execution outcomes, preconditions, and task duration estimates.

Importantly, ACTION APIs are not incompatible with REST. Rather, they extend the REST paradigm by layering a **semantic intent model** atop the traditional HTTP interface. This allows for hybrid implementation strategies where legacy endpoints coexist with verb-oriented aliases, supporting incremental migration and backward compatibility.

## Framing Interaction in Terms of Intent and Context

One of the most significant benefits of the ACTION model is its alignment with how agents simulate behavior. AI agents, particularly those driven by language models or rule-based reasoning, operate on goal-based planning. They require knowledge of what operations are available, what parameters are required, and what result structures are returned.

In a CRUD model, this information is obfuscated by the tight coupling of endpoint names with data structures. In the ACTION model, the API itself serves as a capability surface, an operational map that agents can explore, evaluate, and invoke based on their understanding of task context.

Furthermore, ACTION supports contextual enrichment. Operations can be annotated with metadata such as priority, cost, estimated time, side effects, or resource consumption. This enables agents to make decisions not just on task feasibility, but on task desirability.

The CRUD model, while effective for traditional resource manipulation, lacks the semantic depth and operational granularity required for agentic task execution. It exposes structure, not purpose. It defines data flows, not intent.

The ACTION framework reorients API design around a verb-first, task-oriented model, enabling APIs to function as interfaces for execution, not just access. By embedding intent and semantic clarity into the API surface, ACTION enables AI agents to act with precision, adaptability, and confidence without the need for inter-agent mediation or speculative interpretation.

# 4. ACTION Verb Taxonomy: Capability Language

To enable AI agents to operate effectively within enterprise systems, APIs must transcend resource exposure and declare executable capabilities. Traditional APIs, often aligned with object storage or relational schemas, prioritize data manipulation through CRUD operations (Create, Read, Update, Delete). While efficient for data-centric tasks, these interfaces offer limited guidance for agents executing complex, intent-driven tasks (Allamaraju, 2023). The **ACTION framework**, introduced previously, reorients API design around task-centric operations through six categories: Acquire, Compute, Transact, Integrate, Orchestrate, and Notify. These categories organize APIs by their operational effect, not data structure, providing a semantic scaffold for agent interactions.

Standardizing verbs within these categories is critical to convey capability and affordance. Verbs enable agents to infer endpoint functionality before invocation, reducing ambiguity and enhancing interoperability. In this paper, **Action Verbs** (Hood, 2024) are distinguished from HTTP methods (e.g., GET, POST) and CRUD operations, focusing on semantic intent at the task level to articulate what a service can achieve, not merely what data it exposes (Richardson, 2025).

This section proposes a taxonomy of standardized Action Verbs, organized by ACTION category. This vocabulary is not exhaustive but provides a foundational, extensible framework for semantic API documentation, dynamic capability discovery, and consistent task modeling across services (Hong et al., 2024). By aligning operations with meaningful verbs, APIs become more interpretable and interoperable for AI agents.

## ACTION Examples

### Acquire

*Purpose:* To retrieve, discover, or extract data from internal or external sources, typically with an intent to observe, filter, or assess.

- `search` – Locate data based on query criteria
- `check` – Retrieve or verify the state of a resource
- `scan` – Sweep data sources for conditions or signals
- `discover` – Identify new or related entities
- `extract` – Pull structured or unstructured elements from larger datasets
- `analyze` – Perform observational analysis or pattern detection
- `monitor` – Continuously track a data source for changes
- `retrieve` – Access specific known data assets

### Compute

*Purpose:* To process or transform information into derivative outputs such as summaries, classifications, decisions, or transformations.

- `summarize` – Generate condensed representations of source material
- `validate` – Assess input data against known rules or constraints
- `classify` – Assign data to known categories
- `calculate` – Perform numeric or logical operations
- `predict` – Estimate future states based on models
- `evaluate` – Compare against benchmarks, rules, or standards
- `translate` – Convert between languages or formats
- `rank` – Order data based on defined criteria
- `filter` – Exclude or include data based on logic

### Transact

*Purpose:* To commit or perform operations that result in state change, record persistence, or completion of an external action.

- `book` – Reserve a resource or time
- `purchase` – Complete a commercial transaction
- `register` – Enroll an entity or user in a process or system
- `cancel` – Revoke or reverse a scheduled transaction
- `submit` – Provide a request, application, or form for processing
- `authorize` – Approve permissions or credentials
- `sign` – Digitally or physically confirm agreement or execution
- `transfer` – Move assets, ownership, or records

### Integrate

*Purpose:* To connect, synchronize, or unify data, services, or logic across systems or silos.

- `merge` – Combine data or entities
- `sync` – Align records across systems
- `link` – Associate entities for tracking or logic
- `map` – Define relationships between structures or datasets
- `connect` – Establish a relationship or conduit between systems



- `import` – Bring external data into a local context
- `embed` – Insert one component or dataset into another

### Orchestrate

*Purpose:* To coordinate sequences of tasks, workflows, retries, or conditional logic across time, systems, or agents.

- `schedule` – Define time-based execution of tasks
- `chain` – Link sequential actions for composite execution
- `batch` – Group tasks for bulk execution
- `retry` – Reattempt failed or incomplete tasks
- `delegate` – Assign task execution to another entity
- `escalate` – Elevate priority or reroute a task due to failure or exception
- `pause` – Temporarily halt a process or action
- `resume` – Restart a paused or deferred task

### Notify

*Purpose:* To generate or deliver signals, messages, or outputs to systems, users, or other agents.

- `notify` – Inform a recipient of a state change or event
- `alert` – Send a high-priority or time-sensitive message
- `broadcast` – Disseminate information to multiple recipients
- `report` – Generate a structured summary or update
- `reply` – Respond to an incoming request or message
- `log` – Persist information for auditing or future reference
- `publish` – Make content or results available to a broader audience

## Advantages of Structuring by ACTION

- **Thematic consistency**: Reinforces ACTION as both a design philosophy and implementation framework.

- **Ease of mapping**: Developers and agents can easily associate API functions with their domain category.



- **Extensibility**: Each category can grow with domain-specific sub-verbs (e.g., `reconcile` under Compute for finance, or `triage` under Orchestrate for healthcare).

- **Clarity in tooling**: Enables standardized documentation and metadata generation, grouped by operational domain.

## Building a Vocabulary of Action for Intelligent Systems

The ACTION verb taxonomy provides a structured lexicon of operational intent, organized under the six ACTION pillars. This vocabulary offers a semantically rich, discoverable, and machine-readable foundation for APIs, mapping the capabilities agents require in modern service ecosystems. While not exhaustive, the verbs are generalizable, allowing extensions like `TRIAGE` (healthcare) or `RECONCILE` (finance) under relevant categories to support domain-specific needs.

This taxonomy enables agent capability discovery and automated API composition (Hood, 2025). By tagging interfaces with clear verbs, agents can discern operational intent, reducing reliance on prompt engineering and improving execution reliability. Semantic affordances also enhance orchestration logic, making workflows more resilient (Hong et al., 2024).

Subsequent sections will explore implementing these verbs, documenting them via OpenAPI extensions, and integrating them into the AgenticAPI Specification for next-generation intelligent interfaces.

# 5. Designing APIs for Agents, Not Just Humans

Historically, API design has been human-centered, optimized for consumption by developers who interpret documentation, experiment with payloads, and manually compose workflows. While this model has proven effective for traditional client-server systems, it poses structural limitations for the new generation of AI agents that are expected to interpret, invoke, and chain operations with minimal human intervention.

Agentic systems require APIs that are not only syntactically valid but also **semantically interpretable**. These APIs must expose both capabilities and context to allow agents to reason over options, select appropriate operations, and execute tasks with predictable outcomes. Transitioning from human-first to agent-ready design involves rethinking verb mapping, input/output structures, and affordance communication.

## Mapping Action Verbs to HTTP Methods and Resource Routes

HTTP's limited verb set (GET, POST, PUT, PATCH, DELETE) can serve as a transport layer for richer action semantics within the ACTION framework. Intent is conveyed not through HTTP methods but via explicit verb-oriented resource paths and payloads.

For example:

`POST /recommendations` becomes `RECOMMEND /products?user=123&budget=500`
`GET /summaries` is restructured as `SUMMARIZE /document?id=abc123`

This approach requires defining action verbs as first-class path segments, either directly (e.g., `/ACTION/target`) or as clearly annotated metadata within OpenAPI specifications. These routes should include contextual parameters that signal the intent and operational constraints of the action, enabling agents to infer preconditions and desired outcomes.

## Semantically Rich Schemas and Input Structures

Agents rely on schema metadata not only to format requests but also to understand the conceptual function of a given operation. Therefore, parameter definitions must move beyond basic data types and include:

- **Descriptive annotations**: Clarify purpose, unit, and behavioral implications
- **Enumerated values**: Limit ambiguity and define valid states
- **Examples**: Provide canonical use cases to support model inference



- **Contextual modifiers**: Allow for task customization (e.g., verbosity, priority, output style)

Payloads should be shaped to express intent modifiers as clearly as core data. For instance, a `RECOMMEND` action may include optional fields like `context`, `goal`, or `restrictions` that influence task logic. Such schemas make actions expressive and controllable, aligning with agent reasoning patterns (Hong et al., 2024).

## Chaining and Orchestrating Workflows

Agent-friendly APIs must support complex tasks that span multiple operations, a critical aspect of Agent Experience (AX). To enable chaining and orchestration, AgenticAPI provides predictable naming (e.g., `SCHEDULE`, `RESCHEDULE`), status metadata, and retry mechanisms, aligning with the **Orchestrate** category's focus on workflow coordination. Aggregated endpoints like `BATCH /actions` streamline multi-step processes by combining operations, reducing latency and enhancing usability. These design patterns ensure agents can execute tasks efficiently without external orchestration, as detailed in the implementation blueprint (Section 7).

## Task-First Authentication and Permissioning

Agentic systems often operate on behalf of users or other systems, requiring dynamic authentication contexts. A task-first API must therefore:

- Support delegated authorization (e.g., OAuth 2.0 scopes aligned with specific verbs).
- Allow for capability scoping at the action level (e.g., "Can this agent ANALYZE but not PUBLISH?").
- Use auditable tokens that encode not only identity but also task context and permissions.

These requirements point to a future where least-privilege execution reduces security risks while supporting fine-grained task access. AgenticAPI proposes a task-oriented authentication model aligned with these principles. Building on OAuth 2.0, the design ties scopes directly to specific actions (e.g., `scope: summarize_document`), enabling agents to operate within clearly authorized task boundaries. Unlike protocol-based approaches such as MCP's OAuth 2.1, which emphasize tool-level access, the AgenticAPI model introduces the concept of encoding task context within auditable tokens—enhancing precision, traceability, and safety in agent-driven workflows.

## Predictable Response and Error Patterns

Agents require consistent, machine-parsable responses to avoid reliance on ad hoc error messages. Responses must:

- Be uniform across verbs and implementations.
- Include standardized keys (e.g., *status*, *result*, *next_action*, *errors*).
- Provide descriptive remediation options for automated fallback.
- Support extensible metadata (e.g., latency, costs, human intervention needs).

Error responses should not merely report failure but indicate remediation options, enabling agents to adapt their strategy or escalate appropriately. Standardized responses eliminate quirks by enforcing consistent schemas and documenting edge cases via `x-agent-hints`, enhancing agent reliability.

## Architecting APIs for Agentic Operation

Agentic API design shifts from resource manipulation to task specification. The ACTION framework provides a semantic foundation, but implementation requires embedding intent, context, and execution patterns into the API interface. By mapping verbs to paths, enriching schemas with metadata, and standardizing orchestration, developers create interfaces that are human-readable and machine-operational. Task-scoped authentication and predictable responses further enable intelligent execution.

The next section extends this philosophy into formal documentation, adapting OpenAPI to represent actions, preconditions, and execution semantics within the AgenticAPI Specification.

# 6. OpenAPI + AgenticAPI

The OpenAPI Specification (OAS) is the de facto standard for describing RESTful APIs, enabling documentation, client libraries, and testing frameworks from machine-readable contracts (OpenAPI Initiative, 2023). However, OAS focuses on structural descriptions, including endpoints, parameters, and data formats, rather than semantic intent. For AI agents, which cannot infer purpose from naming conventions or ambiguous documentation, this limitation hinders task execution. Agents require explicit declarations of actions, contexts, and outcomes. The AgenticAPI Specification extends OpenAPI to prioritize intent expression, shifting API discovery from listing endpoints to identifying actionable capabilities.

As agents become more capable of task execution, the question arises: can AI agents negotiate API contracts dynamically? Traditional API design assumes predefined schemas and static interfaces. However, agent-driven workflows often involve ephemeral needs, dynamic tooling, and context-specific actions. Alternative models such as the speculative FLEX design pattern (Hood, 2025) explore how APIs might adapt in real time, not only by spinning up ephemeral interfaces when needed, but also by enabling nested and connected actions within a single transactional scope.

In this model, an agent might initiate a top-level request that internally triggers multiple sub-actions, all composed and resolved based on intent. The interface is no longer a fixed surface; it becomes an active workflow scaffold. AgenticAPI supports this evolution by embedding intent directly into the interface layer, enabling systems to move from static contract binding to real-time capability discovery, composition, and execution.

## Extending OpenAPI to Describe Actions

AgenticAPI augments OpenAPI with structured extensions to describe actions, not just endpoints or methods. Operations are annotated with verb semantics tied to the six ACTION categories: Acquire, Compute, Transact, Integrate, Orchestrate, and Notify. Each operation object (e.g., `paths["/summarize"].post`) includes a custom `x-action` block with:

- `action_verb:` A machine-readable semantic label for the task (e.g., summarize, authorize, schedule).
- `action_category:` One of the six primitives defined by the ACTION model: Acquire, Compute, Transact, Integrate, Orchestrate, Notify.
- `intent_description:` A concise summary of the task's purpose and its intended outcome.

- `contextual_constraints:` Optional conditions that govern when and how the action is available, such as user roles, system states, or environmental variables.
- `preconditions:` Explicit declarations of dependencies or system states that must exist before execution.
- `side_effects:` An outline of any changes the action may trigger, such as state updates, notifications, or downstream impacts.
- `intent_weighting:` Optional metadata for agents to prioritize or evaluate the task, including factors like confidence, cost, urgency, or risk level.
- `output:` A structured description of expected results, including data types, formats, and representations such as JSON, JavaScript, or natural language text (supporting the Adaptive Output principle).

This structure enables agents to reason about ***what the API is capable of doing*** **over** ***what data it can return*****.**

## Adding Semantic Descriptors for Capabilities and Context

In addition to describing operations, API schemas must encode **the conditions and nuances of task execution**. This includes:

- **Task Modifiers**: Optional parameters that adjust execution (e.g., verbosity level, priority, summarization mode)
- **Capability Flags**: Boolean indicators such as `x-supports-batch`, `x-retryable`, `x-human-review-required`
- **Execution Profiles**: Metadata blocks that indicate expected latency, cost, or reliability
- **Agent Guidance**: Optional `x-agent-hints` providing model-specific prompt instructions or fallback mechanisms

These descriptors provide agents with operational knowledge typically buried in documentation or left to developer intuition. By surfacing this metadata in the API spec itself, we improve discoverability, enable real-time decision-making, and reduce the burden of pretraining or fine-tuning on interface behavior.



## From Endpoints to Capabilities

Traditional API discovery mechanisms are name-based and hierarchical. Agents using such mechanisms can retrieve an OpenAPI document, parse its paths, and see which HTTP methods are supported. However, this tells them nothing about **what the service actually does**.

AgenticAPI transforms the discovery process from an endpoint enumeration model to a **capability-oriented model**. Rather than scanning for `GET /products`, an agent can query:

- What actions are available in the `compute` category?
- Are there any `summarize` operations scoped to text documents?
- What actions require authentication with scope `data:write`?
- Which endpoints can notify external systems asynchronously?

This model enables runtime introspection. Agents can select operations dynamically, based on current goals, permissions, and context without relying on brittle endpoint naming or hardcoded integration logic.

## ACTION Metadata as Interface Layer

ACTION metadata provides a semantic bridge between interface and execution. It turns opaque API routes into declarative **capability statements**. This enables a new form of composition: where agents build execution plans based on declared verbs, known contexts, and expected outcomes rather than parsing resource trees and chaining HTTP requests blindly.

Moreover, this metadata layer facilitates:

- **Agent-side planning and optimization** (e.g., comparing latency estimates or precondition trees)
- **Cross-service compatibility checks** (e.g., determining if two services share a common orchestration verb)
- **Tool abstraction** (e.g., allowing different services that implement `summarize` to be interchangeable based on performance)
- **API marketplaces or registries** organized by action categories, not REST paths

This creates an environment where APIs expose what can be done, when, and why.

## Transforming API Contracts into Capability Schemas

OpenAPI excels at documenting resource-based APIs but lacks semantic affordances for agent interaction (OpenAPI Initiative, 2023). AgenticAPI's extensions make intent explicit, categorizing operations with ACTION verbs and enriching definitions with contextual metadata. By shifting from endpoint-focused to action-focused semantics, AgenticAPI transforms OpenAPI into a capability schema for intelligent systems, enabling task-oriented discovery and execution.

## Arazzo and AgenticAPI Synergy

The Arazzo Specification (version 1.0.1, January 2025), introduced by the OpenAPI Initiative, complements AgenticAPI by providing a standardized mechanism to describe deterministic API workflows or sequences of API calls and their dependencies to achieve specific business outcomes. While OpenAPI defines the surface area of individual APIs, Arazzo acts as a "conveyor belt," articulating how multiple APIs interact to complete tasks like user enrollment or flight booking. This synergy enhances AgenticAPI's orchestration capabilities, particularly for AI agents operating in enterprise environments with diverse microservices and SaaS integrations.

Arazzo's workflow objects, which include `sourceDescriptions`, `workflowId`, `inputs`, and `steps`, align closely with AgenticAPI's Orchestrate category and `CHAIN /request` endpoint (Section 7). By integrating Arazzo's structure, AgenticAPI can extend its `DISCOVER /actions` endpoint to include workflow metadata, enabling agents to query multi-API task sequences. For example, a workflow for booking a flight might involve `CHECK /availability`, `BOOK /flight`, and `NOTIFY /user`, with dependencies and success criteria defined explicitly.

**Example: Workflow Discovery with Arazzo Integration**

```
json

{
  "workflows": [
    {
      "workflowId": "bookFlight",
      "summary": "Book a flight and notify user",
      "inputs": {
        "destination": { "type": "string" },
        "date": { "type": "string" },
        "user_id": { "type": "string" }
      },
      "steps": [
        {
```

```
          "stepId": "checkAvailability",
          "x-action": "check",
          "x-category": "acquire",
          "path": "/availability",
          "sourceDescription": {
            "name": "flightAPI",
            "url": "https://api.flightprovider.com/openapi.yaml",
            "type": "openapi"
          },
          "parameters": [
            { "name": "destination", "value": "$inputs.destination" },
            { "name": "date", "value": "$inputs.date" }
          ],
          "successCriteria": { "condition": "$statusCode == 200" }
        },
        {
          "stepId": "bookFlight",
          "x-action": "submit",
          "x-category": "transact",
          "path": "/booking",
          "parameters": [
            { "name": "flight_id", "value":
"$steps.checkAvailability.output.flight_options[0].id" },
            { "name": "user_id", "value": "$inputs.user_id" }
          ],
          "successCriteria": { "condition": "$statusCode == 201" }
        }
      ],
      "scopes_required": ["booking:write"],
      "x-arazzo-version": "1.0.1"
    }
  ]
}
```

**OpenAPI Metadata with Arazzo Extension**:

```
yaml

paths:
  /workflows:
    get:
```

```
      x-action: "discover_workflows"
      x-category: "orchestrate"
      x-arazzo: true
      summary: "Discover available workflows"
      responses:
        '200':
          description: "List of workflows with Arazzo-compatible metadata"
          content:
            application/json:
              schema:
                type: object
                properties:
                  workflows:
                    type: array
                    items:
                      type: object
                      properties:
                        workflowId: { type: string }
                        summary: { type: string }
                        inputs: { type: object }
                        steps: { type: array }
                        scopes_required: { type: array, items: { type:
string } }
                      required: [workflowId, summary, steps]
```

This integration allows AgenticAPI to leverage Arazzo's deterministic workflows while maintaining its task-centric semantics. Arazzo's sourceDescriptions field enhances interoperability by referencing external OpenAPI documents, aligning with AgenticAPI's goal of simplifying multi-API orchestration. The x-arazzo extension signals compatibility, enabling tools to process workflows in software development lifecycle (SDLC) pipelines, supporting automated testing and contract adherence (Section 9). By combining Arazzo's workflow focus with AgenticAPI's contextual intelligence and semantic discoverability, agents can execute complex tasks with greater reliability and scalability, reducing reliance on prompt engineering or external orchestration.



# 7. Implementation Blueprint

Translating the ACTION framework and AgenticAPI Specification from conceptual model to production-ready architecture requires a structured, incremental approach. Organizations with existing RESTful or CRUD-based APIs will face the dual challenge of introducing action-oriented semantics without disrupting current functionality. This section outlines a pragmatic implementation strategy that balances **adoption feasibility**, **semantic integrity**, and **backward compatibility**.

## Migrating from CRUD to ACTION

Enterprise APIs often expose endpoints tied to data models (e.g., entities, records) rather than operational tasks. Transitioning to an ACTION-compliant design involves reframing these as intent-driven actions. Developers can begin with functional mapping: identify user- or system-driven tasks (e.g., "summarize a document," "notify a user"), evaluate whether current endpoints (e.g., `POST /documents`) reflect actions or objects, and redefine them as verb-based operations (e.g., `SUMMARIZE /document`).

The following table illustrates common CRUD-to-**ACTION** mappings:

| CRUD Operation | Category | Verb Example | Semantic Shift |
|---|---|---|---|
| `POST /users` | Transact | `REGISTER /user` | Object creation to enrollment |
| `GET /orders` | Acquire | `SEARCH /orders` | Data listing to discovery |
| `PUT /user/1` | Compute | `VALIDATE /user` | Update to rule-based checking |
| `DELETE /event` | Transact | `CANCEL /event` | Deletion to withdrawal |

These mappings expose task intent, documented with contextual availability (e.g., `VALIDATE /user` requires admin role), aligning with **Contextual Alignment** to prevent invalid operations. Original CRUD endpoints can be aliased (e.g., `POST /users` → `REGISTER /user`), supporting hybrid coexistence for traditional and agentic clients.

## Structuring Payloads and Parameters

**ACTION** endpoints must be self-describing and context-aware, using **Pydantic** schemas in FastAPI for type safety. Input schemas should include:



- **Intent Modifiers**: Values altering execution (e.g., `mode=brief, priority=high`).
- **Context References**: Links to entities or states (e.g., `user_id, session_id`).
- **Execution Hints**: Fields guiding interpretation (e.g., `language=en, format=table`).

For multi-step workflows, payloads may include execution plans (e.g., sub-actions, retry policies). All parameters are documented with OpenAPI extensions (e.g., `x-intent-impact, x-example`) to enable agent reasoning without external documentation.

**Example:** `SUMMARIZE /document`

```
python

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI()

class SummarizeQuery(BaseModel):
    document_id: str
    format: str = "text"  # text, bullets
    max_words: int = 50
    style: str = "neutral"  # formal, casual, technical
    output_format: str = "json"  # json, text
    return_raw: bool = False

@app.api_route("/document", methods=["SUMMARIZE"], openapi_extra={"x-action":
"summarize", "x-category": "compute"})
async def summarize_document(query: SummarizeQuery):
    try:
        # Simulated summarization logic
        summary = "Comic books evolve with digital platforms, diverse creators,
hybrid formats, and cultural impact."
        return {"summary": summary, "title": "Document Title", "output_format":
query.output_format}
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Summarization failed:
{str(e)}")
```

**Request**:

```
{
  "document_id": "c1",
  "format": "text",
  "max_words": 20,
```



```
  "style": "neutral",
  "output_format": "json"
}
```

**Response**:

```
{
  "summary": "Comic books evolve with digital platforms, diverse creators, hybrid
formats, and cultural impact.",
  "title": "Comic Book Evolution"
}
```

**Response** (Conversational, `?output_format=text`):

```
text

Comic books have transformed through digital platforms, empowering diverse
creators and blending print with innovative formats for cultural impact.
```

## Input with Semantic Discoverability

Semantic discoverability enables agents to understand and invoke API capabilities without relying on natural language prompts or reverse engineering. AgenticAPI achieves this by exposing actions as discoverable endpoints, allowing agents to query available operations dynamically, as outlined in the capability-oriented discovery model. For example, agents can explore what actions an API supports, their categories, and execution constraints, aligning with the Agent Experience (AX) demand for machine-first consumption.

Consider a discovery endpoint that lists available actions:

**Example:** `DISCOVER /actions`

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "actions": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "action_verb": { "type": "string", "enum": ["summarize", "book",
"notify"] },
          "category": { "type": "string", "enum": ["Compute", "Transact", "Notify"]
```

 

```
},
          "path": { "type": "string", "example": "/document" },
          "scopes_required": { "type": "array", "items": { "type": "string" },
"example": ["data:read"] },
          "preconditions": { "type": "string", "example": "document_id exists" }
        },
        "required": ["action_verb", "category", "path"]
      }
    }
  },
  "required": ["actions"]
}
```

**Example:** `DISCOVER /workflows`

To support deterministic workflows, the `DISCOVER /workflows` endpoint returns metadata about available workflows, including their steps, inputs, and source APIs. This leverages Arazzo's workflow object structure to ensure agents can execute complex tasks with clear dependencies.

```
json

{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "workflows": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "workflowId": { "type": "string", "example": "bookFlight" },
          "summary": { "type": "string", "example": "Book a flight and notify user"
},
          "inputs": {
            "type": "object",
            "properties": {
              "destination": { "type": "string" },
              "date": { "type": "string" },
              "user_id": { "type": "string" }
            }
          },
          "steps": {
            "type": "array",
```



```
            "items": {
              "type": "object",
              "properties": {
                "stepId": { "type": "string", "example": "checkAvailability" },
                "action_verb": { "type": "string", "example": "CHECK" },
                "path": { "type": "string", "example": "/availability" },
                "sourceDescription": {
                  "type": "object",
                  "properties": {
                    "name": { "type": "string", "example": "flightAPI" },
                    "url": { "type": "string", "example":
"https://api.flightprovider.com/openapi.yaml" },
                    "type": { "type": "string", "enum": ["openapi"] }
                  }
                }
              }
            },
            "scopes_required": { "type": "array", "items": { "type": "string" },
"example": ["booking:write"] }
          },
          "required": ["workflowId", "summary", "steps"]
        }
      }
    },
    "required": ["workflows"]
}
```

**OpenAPI Metadata**:

```yaml
yaml

paths:
  /workflows:
    get:
      x-action: "discover_workflows"
      x-category: "orchestrate"
      summary: "Discover available workflows"
      responses:
        '200':
          description: "List of workflows"
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/WorkflowDiscovery'
```



```
components:
  schemas:
    WorkflowDiscovery:
      type: object
      properties:
        workflows:
          type: array
          items:
            type: object
            properties:
              workflowId: { type: string }
              summary: { type: string }
              inputs: { type: object }
              steps: { type: array }
              scopes_required: { type: array, items: { type: string } }
            required: [workflowId, summary, steps]
```

This response enables agents to identify executable actions and workflows along with their semantic context, reducing dependency on external documentation. The `sourceDescription` field ensures interoperability with external APIs, aligning with enterprise sandbox environments.

To ensure payloads are unambiguous, AgenticAPI minimizes optional fields and annotates them with x-intent-impact in OpenAPI schemas, clarifying how parameters alter outcomes (e.g., destination in `CHECK /availability` filters flight options). This reinforces Semantic Discoverability, enabling agents to process inputs reliably without assumptions.

To ensure payloads are unambiguous, AgenticAPI carefully designs input parameters. To prevent agent misinterpretation, AgenticAPI minimizes optional fields and annotates them with `x-intent-impact` in OpenAPI schemas, clarifying how they alter outcomes (e.g., `format=bullet` changes response structure). This approach reinforces Semantic Discoverability, enabling agents to process inputs reliably without assumptions.

## Contextual Intelligence: Dynamically Adapting to Intent

AgenticAPI's contextual intelligence empowers AI agents to interpret and adapt the intent behind versatile verbs like `CHECK`, which can represent diverse tasks—checking weather, availability, flight status, grocery lists, schedules, KPIs, or due dates. Traditional APIs struggle with such ambiguity, forcing agents to rely on external logic or documentation to discern meaning. AgenticAPI addresses this by embedding intent into the API layer, using semantic metadata to dynamically adjust `CHECK` based on context, ensuring precise, relevant responses.

Through the `DISCOVER /actions` endpoint, agents query available actions and receive metadata clarifying what **CHECK** means in each context. For example, `CHECK /weather`



retrieves forecasts, while `CHECK /availability` confirms meeting slots. This leverages the ACTION taxonomy (Acquire, Compute, Transact, etc.), aligning with Semantic Discoverability to make intent machine-readable. AI-driven contextual analysis minimizes misinterpretation, enabling agents to execute tasks without hardcoded assumptions, enhancing the Agent Experience (AX).

This approach supports natural language semantics, allowing users and agents to maximize process efficiency across domains like finance, healthcare, or logistics. By reducing reliance on external orchestration, AgenticAPI streamlines automation, lowers error rates, and scales effortlessly, as demonstrated in proof-of-concepts for tasks like scheduling.

**Example:** `DISCOVER /actions`


```
{
  "actions": [
    {
      "action_verb": "CHECK",
      "category": "Acquire",
      "path": "/weather",
      "description": "Retrieve weather for a location",
      "inputs": ["location"],
      "scopes_required": ["weather:read"]
    },
    {
      "action_verb": "CHECK",
      "category": "Acquire",
      "path": "/availability",
      "description": "Check meeting slot availability",
      "inputs": ["date", "participants"],
      "scopes_required": ["calendar:read"]
    },
    {
      "action_verb": "CHECK",
      "category": "Transact",
      "path": "/flight_status",
      "description": "Verify flight status",
      "inputs": ["flight_number"],
      "scopes_required": ["flight:read"]
    }
  ]
}
```


**OpenAPI Metadata**



```yaml
paths:
  /{context}:
    check:
      x-action: "check"
      x-category: "acquire"
      summary: "Check context-specific information"
      parameters:
        - name: context
          in: path
          required: true
          schema:
            type: string
            enum: [weather, availability, flight_status]
          x-intent-impact: "Defines the domain of the check action"
```

This metadata ensures agents dynamically adapt `CHECK` to the intended task, making AgenticAPI a foundation for intent-driven, scalable automation.

## Standardized Output with Execution Clarity and Adaptive Representation

Responses must reflect both outcome and agent usability. For example:

```json
{
  "status": "completed",
  "summary": "Comic books evolve with digital platforms, diverse creators, hybrid formats, and cultural impact.",
  "title": "Comic Book Evolution",
  "output_format": "json",
  "confidence": 0.92
}
```

This model supports:

- **Execution Clarity**: `status` and `next_action` clarify outcomes and follow-ups.
- **Intent Weighting**: `confidence` expresses internal certainty
- **Adaptive Output**: `output_type` enables formats like text, markdown, or JSON.



## Integrating Compatibility and Extensibility

**AgenticAPI** extends REST conventions without breaking compatibility. An aliasing strategy ensures coexistence:

```yaml
paths:
  /document:
    post:
      summary: "Summarize a document (POST)"
      operationId: summarizeDocumentPost
      x-alias-for: "SUMMARIZE /document"
  /document:
    summarize:
      x-action: "summarize"
      x-category: "compute"
      summary: "Summarize a document"
      operationId: summarizeDocument
```

Swagger UI displays both paths, while agents use `x-action` metadata, supporting **Compatibility and Extensibility** (Section 6) and PoC's integration complexity goal.

## Embedding Intent Weighting and Sensitivity

Agents must often select among multiple similar options. Adding **intent qualifiers** helps resolve ambiguity:

```yaml
x-intent-weighting:
  priority: "high"
  cost_estimate: 0.004  # in USD
  risk_profile: "low"
```

This enables agents to compare actions (e.g., `SUMMARIZE` vs. `TRANSLATE`), optimizing based on cost or urgency (Wooldridge & Jennings, 1995), per the **Intent Weighting** principle.

## Orchestrating Complex Workflows

Real-world agent tasks often span multiple operations, requiring APIs to support chaining, recursion, and adaptive workflows. To enable agents to orchestrate sequences based on

success, failure, or context, AgenticAPI provides standardized patterns and implementation mechanisms, aligned with the **Orchestrate** category (Appendix A). APIs should:

- Provide unique identifiers for each action instance to ensure traceability.
- Return linkable references to follow-up operations (e.g., a `BOOK` action linking to `MODIFY`).
- Include status metadata to clarify whether additional steps are required.
- Use predictable naming conventions (e.g., `SCHEDULE, RESCHEDULE, CANCEL`) for intuitive action relationships.
- Expose retry policies and fallback actions (e.g., `RETRY /action`, `ESCALATE /workflow`) to handle failures gracefully.

These patterns enable agents to construct adaptive workflows, retry failed steps, or continue partially completed processes without external orchestration systems.

**Example:** `CHAIN /request`

```
python

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import List, Dict, Any

app = FastAPI()

class ChainStep(BaseModel):
    verb: str
    path: str
    params: Dict[str, Any]

class ChainRequest(BaseModel):
    chain: List[ChainStep]

@app.post("/chain", openapi_extra={"x-action": "chain", "x-category":
"orchestrate"})
async def chain_verbs(chain_request: ChainRequest):
    results = []
    for step in chain_request.chain:
        verb = step.verb.upper()
        try:
            # Simulated verb processing
            result = {"status": "completed", "output": f"{verb} executed"}
            results.append({"step": step.model_dump(), "result": result})
```



```
        except Exception as e:
            results.append({"step": step.model_dump(), "error": str(e)})
    return {"results": results}
```

**Request:**

```
{
  "chain": [
    {
      "verb": "SEARCH",
      "path": "/orders",
      "params": {"query": "pending", "output_format": "json"}
    },
    {
      "verb": "NOTIFY",
      "path": "/notify",
      "params": {
        "recipient": "user@example.com",
        "message": "Order processed",
        "channel": "email",
        "output_format": "json"
      }
    }
  ]
}
```

**Response:**

```
{
  "results": [
    {
      "step": {
        "verb": "SEARCH",
        "path": "/orders",
        "params": {"query": "pending", "output_format": "json"}
      },
      "result": {
        "status": "completed",
        "output": "SEARCH executed"
      }
    },
    {
      "step": {
        "verb": "NOTIFY",
        "path": "/notify",
        "params": {
```



```
          "recipient": "user@example.com",
          "message": "Order processed",
          "channel": "email",
          "output_format": "json"
        }
      },
      "result": {
        "status": "completed",
        "output": "NOTIFY executed"
      }
    }
  ]
}
```

This schema supports looping, branching, retries, and fallbacks, documented with `x-workflow` and `x-dependency`. Real-time adaptation adjusts workflows dynamically, enhancing flexibility (Section 5).

**Example: Workflow Specification for Flight Booking**

To support complex tasks across multiple APIs, AgenticAPI introduces a Workflow Specification Object for payloads, defining deterministic sequences of actions. This extends the `CHAIN /request` endpoint to incorporate Arazzo's structure, ensuring agents execute workflows like flight booking with clear dependencies.

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import List, Dict, Any

app = FastAPI()

class WorkflowStep(BaseModel):
    stepId: str
    action_verb: str
    path: str
    parameters: List[Dict[str, Any]]
    successCriteria: Dict[str, Any]

class WorkflowSpec(BaseModel):
    workflowId: str
    inputs: Dict[str, Any]
```

 

```
    steps: List[WorkflowStep]
    sourceDescriptions: List[Dict[str, str]]

@app.post("/workflow/{workflowId}", openapi_extra={"x-action": "execute_workflow",
"x-category": "orchestrate"})
async def execute_workflow(workflowId: str, workflow: WorkflowSpec):
    try:
        results = []
        for step in workflow.steps:
            # Simulated step processing
            result = {"status": "completed", "output": f"{step.action_verb}
executed"}
            results.append({"step": step.dict(), "result": result})
        return {
            "workflowId": workflowId,
            "status": "completed",
            "results": results
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Workflow execution failed:
{str(e)}")
```

**Request**:

```
json

{
  "workflowId": "bookFlight",
  "inputs": {
    "destination": "San Francisco",
    "date": "2025-06-01",
    "user_id": "user123"
  },
  "sourceDescriptions": [
    {
      "name": "flightAPI",
      "url": "https://api.flightprovider.com/openapi.yaml",
      "type": "openapi"
    }
  ],
  "steps": [
    {
      "stepId": "checkAvailability",
      "action_verb": "CHECK",
      "path": "/availability",
```



```
      "parameters": [
        { "name": "destination", "value": "$inputs.destination" },
        { "name": "date", "value": "$inputs.date" }
      ],
      "successCriteria": { "condition": "$statusCode == 200" }
    },
    {
      "stepId": "bookFlight",
      "action_verb": "BOOK",
      "path": "/booking",
      "parameters": [
        { "name": "flight_id", "value":
"$steps.checkAvailability.outputs.flight_options[0].id" },
        { "name": "user_id", "value": "$inputs.user_id" }
      ],
      "successCriteria": { "condition": "$statusCode == 201" }
    }
  ]
}
```

**Response**:

```
json

{
  "workflowId": "bookFlight",
  "status": "completed",
  "results": [
    {
      "step": {
        "stepId": "checkAvailability",
        "action_verb": "CHECK",
        "path": "/availability",
        "parameters": [
          { "name": "destination", "value": "San Francisco" },
          { "name": "date", "value": "2025-06-01" }
        ],
        "successCriteria": { "condition": "$statusCode == 200" }
      },
      "result": { "status": "completed", "output": "CHECK executed" }
    },
    {
      "step": {
        "stepId": "bookFlight",
        "action_verb": "BOOK",
```



```
        "path": "/booking",
        "parameters": [
          { "name": "flight_id", "value":
"$steps.checkAvailability.outputs.flight_options[0].id" },
          { "name": "user_id", "value": "user123" }
        ],
        "successCriteria": { "condition": "$statusCode == 201" }
      },
      "result": { "status": "completed", "output": "BOOK executed" }
    }
  ]
}
```

**OpenAPI Metadata**:

```yaml
paths:
  /workflow/{workflowId}:
    post:
      x-action: "execute_workflow"
      x-category: "orchestrate"
      summary: "Execute a defined workflow"
      parameters:
        - name: workflowId
          in: path
          required: true
          schema:
            type: string
          x-intent-impact: "Identifies the workflow to execute"
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/WorkflowSpec'
components:
  schemas:
    WorkflowSpec:
      type: object
      properties:
        workflowId: { type: string }
        inputs: { type: object }
        steps: { type: array }
        sourceDescriptions: { type: array }
      required: [workflowId, steps]
```



This payload structure ensures agents can execute multi-API workflows with clear intent, leveraging Arazzo's deterministic approach while maintaining AgenticAPI's task-centric semantics.

## Test Mode

AgenticAPI enhances safety and reliability by supporting a test mode, allowing agents to simulate actions without committing changes. Aligned with the **Compute** category's focus on evaluation (e.g., `VALIDATE`, `EVALUATE`), endpoints like `TEST /action` return anticipated outcomes, side effects, and validation checks, enabling agents to assess task feasibility before execution. This capability, rooted in the **Execution Clarity** principle, is critical for high-stakes domains like finance (e.g., testing a payment authorization) and healthcare (e.g., simulating patient triage).
**Example:** `TEST /action`

```
python

from fastapi import FastAPI
from pydantic import BaseModel
from typing import Dict, Any

app = FastAPI()

class TestRequest(BaseModel):
    verb: str
    path: str
    payload: Dict[str, Any]

@app.post("/action", openapi_extra={"x-action": "test", "x-category":
"compute"})
async def test_action(test: TestRequest):
    return {
        "verb": test.verb,
        "path": test.path,
        "status": "valid",
        "outcome": {"example": "Simulated result"},
        "side_effects": ["none"],
        "validation_checks": {"preconditions_met": True}
    }
```



Test mode reduces errors by enabling agents to verify preconditions and anticipate impacts, fostering trust in agent-driven automation. For example, a financial agent can test a `TRANSFER /funds` action to confirm compliance, while a healthcare agent can simulate `TRIAGE /patient` to validate protocol adherence. Integrated with workflows (Section 7), test mode supports pre-execution checks for orchestrated tasks, enhancing AX by ensuring precision and scalability in dynamic environments.

## Versioning and Compatibility Considerations

To maintain system stability, ACTION-based additions should be treated as non-breaking enhancements. Use OpenAPI's versioning mechanisms (e.g., `info.version`, `x-api-version`) to differentiate action-enriched specs from legacy ones.

When possible:

- Maintain consistent identifiers across old and new paths.
- Signal ACTION readiness via capability flags (e.g., `x-action-supported: true`).
- Allow clients to opt in to ACTION routing via headers or discovery metadata.

This ensures a smooth transition path and minimizes disruption to existing workflows.

## Operationalizing Intent

This revised implementation blueprint demonstrates how each feature of the AgenticAPI Specification can be operationalized in practice. By aligning endpoint design with ACTION categories, exposing semantic metadata, supporting contextual execution logic, and designing for nested orchestration flows, developers can build APIs that agents can not only call but understand, evaluate, and adapt to.

Crucially, the framework is designed for incremental adoption. Organizations can alias existing endpoints, enrich OpenAPI specs with metadata, and phase in advanced features such as intent weighting and adaptive output formats without disrupting existing clients.

 

# 8. Comparative Analysis

As organizations evaluate how to modernize their interfaces for AI agent consumption, it is essential to assess the strengths and limitations of prevailing API paradigms. Each model, CRUD, GraphQL, REST, and protocol-based frameworks like MCP offers distinct benefits and trade-offs. The ACTION framework, as defined in the AgenticAPI Specification, introduces an intent-first, task-oriented design model that aligns with the operational requirements of intelligent systems.

This section provides a structured comparison of ACTION relative to alternative approaches, across dimensions of semantic clarity, orchestration support, complexity, maintainability, and agent usability.

## ACTION vs. CRUD

The CRUD model, designed for low-level data manipulation (Create, Read, Update, Delete), prioritizes structure over purpose (Fielding, 2000). While effective for database abstraction and developer control, CRUD lacks operational intent, increasing cognitive burden for agents inferring endpoint functions. For example, an endpoint like `POST /reports` could imply `GENERATE`, `SUBMIT`, or `APPROVE`, causing ambiguity. In contrast, ACTION employs semantic verbs (e.g., `SUMMARIZE`, `BOOK`, `RECOMMEND`) to encode goals, reducing processing complexity for automated systems. Under ACTION, each task is distinct and discoverable.

Moreover, CRUD offers no built-in orchestration semantics, forcing agents to construct multi-step tasks externally. ACTION supports execution chaining, retry semantics, and contextual awareness, enabling workflow assembly for advanced agents. These features make ACTION more suitable for agent-driven systems.

## ACTION vs. GraphQL

GraphQL shifts from fixed endpoints to client-defined queries, emphasizing data flexibility over execution semantics (GraphQL, 2023). It reduces over-fetching and enhances performance for front-end applications but does not model tasks. GraphQL mutations are often generic, lacking clear intent for agent-based decision-making (Hartig & Pérez, 2017). For instance, a mutation named processOrder may obscure multiple side effects, complicating agent evaluation.

Conversely, ACTION defines capabilities as verbs within functional domains (e.g., Acquire, Compute, Transact), allowing agents to reason about actions, constraints, and execution paths (Verborgh et al., 2016). While GraphQL's type system is expressive, it prioritizes data schema over intent. ACTION extends OpenAPI schemas with contextual metadata, enabling agents to assess task suitability, outcomes, and side effects before invocation.

## ACTION vs. Traditional REST

Traditional RESTful APIs organize routes around resources, using URL paths and HTTP verbs to imply capability. While REST brought order to early web APIs, it offers limited **semantic signaling** to automated clients. Endpoints such as `PUT /status` or `POST /payment` require documentation parsing or hardcoded interpretation to determine what effect they perform.

ACTION augments this model by elevating verbs to **first-class entities**. For example, `NOTIFY /user`, `AUTHORIZE /account`, and `SCHEDULE /task` make action and intent explicit. Additionally, ACTION introduces metadata such as `preconditions`, `outcomes`, and `confidence`, allowing agents to plan and adapt, functionality that REST endpoints do not natively support.

Moreover, REST does not support multi-step task orchestration, error remediation pathways, or fallback alternatives at the API layer. These must be constructed in external systems, increasing maintenance complexity and agent-side burden.

## ACTION vs. MCP / A2A Protocols

Proposals like Model Context Protocol (MCP) and Agent-to-Agent (A2A) communication aim to create a protocol layer for agent coordination, enabling discovery, negotiation, and task delegation between models. While the concept has theoretical merit, these approaches introduce unnecessary complexity without addressing the core issue: how agents interact with actions and data effectively.

MCP does not replace APIs. It adds an additional intermediation layer that increases latency, creates new points of failure, complicates versioning, and makes security and observability harder to manage. It also assumes that agent-to-agent collaboration is essential, when in most practical workflows, an agent can complete the task through a direct call to a well-designed API.

AgenticAPI offers a more effective solution. Rather than adding protocol layers, it focuses on improving API design by using clear operational verbs and context-aware contracts. This allows agents to invoke APIs directly, receive structured outputs, and complete complex tasks without relying on external coordination mechanisms.

The goal is not to eliminate APIs but to evolve them into intelligent interfaces that agents can understand and act on independently. AgenticAPI simplifies the system while preserving the benefits of secure, scalable, and interpretable integration.

 

## Core Comparison: MCP vs AgenticAPI

| Category | MCP | AgenticAPI |
|---|---|---|
| **Interface** | Local plugin server or YAML-defined manifest | API-native contract (verbs with structured logic) |
| **Execution Model** | Shell/process-based, often insecure, DIY | HTTP-based, secure, governed, and observable |
| **Versioning** | Largely ad-hoc | Potential for formal versioning, contracts, governance |
| **Use Case Fit** | Chat-UI plugins, hobbyist workflows | Structured, scalable agent-to-service execution |
| **Security** | Poor by default, requires user hardening | Built for enterprise integration and observability |
| **Scalability** | Local and brittle | Centralized and standard-compliant |
| **Developer UX** | YAML hell and untyped JSON | IDE-discoverable, verb-based API design |
| **Data vs. Action** | Mostly passthrough, requires extra logic | Can embed intelligence or return clean data for agent use |

## Strategic Differentiation from MCP

| Positioning Claim | Supporting Argument |
|---|---|
| *AgenticAPI is not a plugin layer* | It retains API integrity, versioning, and observability. |
| *It is not brittle or local* | No shelling out, no unvetted code execution. |
| *It is semantically richer than CRUD* | Action verbs encode task intent (`SUMMARIZE`, `TRANSLATE`, `DECIDE`). |
| *It scales across agents, not just chat UIs* | Designed for interoperable agents and services, not just user assistants. |
| *It embeds security from day one* | MCPs assume trust; AgenticAPI enforces it. |

## Comparative Summary Table

| Model | Intent Clarity | Agent Usability | Orchestration Support | Protocol Complexity | Adoption Cost |
|---|---|---|---|---|---|
| CRUD | Low | Low | None | Low | Minimal |
| GraphQL | Medium | Low | Limited | Medium | High |
| REST | Medium | Medium | External only | Low | Minimal |
| MCP / A2A | High | Medium | High | Very High | Very High |
| **ACTION** | **High** | **High** | **Native** | **Low** | **Moderate** |

# 9. Organizational Impact

The introduction of the ACTION framework and the AgenticAPI Specification represents more than a shift in interface mechanics. It marks a transformation in how organizations conceive, build, and manage their integration layer. As APIs evolve from data access tools to action-oriented capabilities, development teams, system architects, and business stakeholders must realign their processes, standards, and expectations to fully realize the benefits of intent-driven design.

This section examines the practical implications of adopting ACTION for API-producing teams and the broader strategic benefits for organizations pursuing intelligent automation at scale.

## Impact on API Teams

### Updated Design Workflows

ACTION requires API teams to transition from resource modeling to task modeling. Rather than begin with database tables or object schemas, design processes should originate from goal decomposition: What tasks do users or agents need to perform? What operations need to be exposed to satisfy those tasks in a predictable, executable manner?

This shift encourages cross-functional collaboration between product managers, domain experts, and API designers (Nylén & Holmström, 2015). Task design becomes a shared language that spans functional and technical disciplines. As a result, teams must adopt:

- **Verb-driven design templates**
- **Task-specific contract definition**
- **Execution context mapping** (preconditions, expected outcomes, side effects)

Teams may also need to define domain-specific verb libraries, extendable from the ACTION root taxonomy, to support vertical alignment (e.g., `reconcile`, `triage`, `route`).

### New Developer Onboarding Patterns

Developer onboarding will evolve to include **action vocabulary fluency**, not just endpoint familiarity. With ACTION-compliant APIs, documentation focuses on:

- What each service **does** (actions)
- Under what **conditions** it operates
- How agents and developers **invoke**, **chain**, or **remediate** tasks

New developers (human or machine) can be guided through **capability-based discovery** rather than structural path traversal. This reduces cognitive load and accelerates integration readiness.

### Documentation, Testing, and Observability

The AgenticAPI model demands higher semantic rigor in API documentation. Each operation must be annotated with:

- **Intent descriptors** (e.g., `x-action`, `x-category`)
- **Precondition logic**
- **Execution metadata** (e.g., latency, confidence thresholds, required scopes)

Documentation tooling such as Swagger or Stoplight can be extended to render action verbs as navigable units of capability. Additionally, observability systems must be adapted to monitor action-level outcomes, such as success rates, fallback usage, or escalation frequency, metrics often abstracted away in CRUD-centric systems. Platforms like PolyAPI, with its real-time runtime visibility and comprehensive resource cataloging, exemplify how such observability can be achieved (PolyAPI, n.d.).

Testing frameworks will also shift toward task correctness and semantic validation, ensuring not only syntactic integrity but also appropriate execution in real-world contexts. For example, contract tests may assert that a `RECOMMEND /products` action excludes banned SKUs or respects pricing filters under specific conditions.

## Strategic Business Value

### Shorter Development Cycles

By exposing semantically rich, self-describing APIs, development teams reduce the need for extensive client-side interpretation or custom logic scaffolding (Wang & McLarty, 2021). Agents and developers can onboard faster, construct task flows more easily, and validate integration behavior in fewer iterations. This leads to:

- Reduced time-to-market for new services
- Faster integration cycles for partners and clients
- Lowered need for platform support or technical debt remediation

### Lower Integration Friction

ACTION reduces the impedance mismatch between what APIs offer and what agents or applications intend to do. This dramatically lowers integration friction, particularly in multi-vendor ecosystems, where semantically aligned verbs reduce the need for brittle API mediation layers or prompt engineering.



Organizations benefit from:

- Easier tool substitution and component reusability
- Better contract alignment with business processes
- Decreased reliance on tribal knowledge or internal specialists

### More Intelligent Automation at Lower Cost

Perhaps most critically, ACTION enables intelligent automation to be deployed without requiring agents to agent or protocol-based negotiation. Agents become more effective with less training, fewer assumptions, and higher execution reliability, yielding:

- Lower automation engineering costs
- Fewer failure modes in complex workflows
- Higher confidence in auditability and governance of automated actions

By clarifying what systems can do and under what conditions, ACTION empowers both human operators and machine agents to act decisively without requiring deep systems knowledge or brittle integration patterns.

## Redesigning the Interface Layer for Scalable Intelligence

Adopting the ACTION framework and AgenticAPI Specification reshapes not only how APIs are consumed, but how they are designed, documented, and deployed. This shift moves APIs from data exposure to capability expression, enabling measurable gains in integration velocity, developer efficiency, and agent-driven task execution.

As AI agents become embedded across enterprise workflows, they introduce new demands on infrastructure, particularly in handling unpredictable, high-frequency, task-based requests. AgenticAPI addresses this by supporting intelligent scalability features such as action-aware rate limiting, dynamic throttling, and usage-based entitlements. Metadata fields like `x-cost-estimate` give agents visibility into resource usage, allowing them to make performance-conscious decisions and avoid unnecessary overload.

By aligning API architecture with agent behavior, AgenticAPI equips organizations to scale intelligent automation without compromising system reliability. In this emerging landscape, APIs that expose clear, intent-driven operations will become foundational infrastructure for adaptive, resilient digital ecosystems.

## API Governance with ACTION

Effective API governance ensures secure, scalable, and compliant agent-driven ecosystems, enhancing AX by providing predictable, reliable interactions. Unlike traditional models reliant on static schemas and coarse-grained controls, AgenticAPI embeds dynamic, granular governance

into the API surface, aligning with **Semantic Discoverability** and **Execution Clarity** principles (Executive Summary) to support enterprise-grade automation.

ACTION verbs (e.g., `AUTHORIZE`, `ESCALATE`, Appendix A) enforce semantic clarity, eliminating API quirks that confuse agents. For example, `NOTIFY /team` ensures consistent behavior, discoverable via `DISCOVER /actions` (Section 7), simplifying policy enforcement for access control and rate limiting across multi-agent systems.

Fine-grained authentication, built on OAuth 2.0 (Section 5), uses task-specific scopes (e.g., `scope: book_flight`) in auditable tokens, supporting forensic analysis and compliance with regulations like GDPR or HIPAA. A financial agent's `TRANSFER /funds`, for instance, is logged to verify authorized access, ensuring least-privilege execution.

Observability metrics, such as `x-success-rate` and `x-escalation-thresholds`, enable real-time monitoring. An audit log schema illustrates this:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "action_id": { "type": "string", "example": "act_789" },
    "verb": { "type": "string", "example": "BOOK" },
    "timestamp": { "type": "string", "format": "date-time", "example":
"2025-06-01T14:00:00Z" },
    "status": { "type": "string", "enum": ["success", "failed"], "example":
"success" },
    "escalation": { "type": "boolean", "example": false }
  },
  "required": ["action_id", "verb", "timestamp", "status"]
}
```

Annotated with `x-audit-log`, this feeds dashboards or machine-learning monitors, adapting workflows proactively (e.g., adjusting `RETRY /action` thresholds). Metrics also track error patterns, enhancing reliability and supporting proactive resolution of agent-driven issues.

Lifecycle governance treats actions as versioned, contract-bound capabilities, ensuring traceability during deprecations or updates while maintaining compatibility and extensibility. Compliance is enforced via role-based access and audit trails, scaling for distributed systems and integrating with test mode for safe validation.

AgenticAPI's governance layer enforces policies, monitors performance, and ensures compliance without stifling innovation, providing a secure foundation for intelligent automation.



# 10. The Future of AI-System Integration

As intelligent agents continue to expand their role across enterprise systems, the interface layer between these agents and underlying services will become a defining factor in overall system effectiveness, adaptability, and security. The AgenticAPI Specification, grounded in the ACTION framework, offers a foundation for building APIs that support this new generation of integrations that are task-oriented, semantically expressive, and natively interpretable by systems.

This section explores the long-term implications and emerging opportunities surrounding the AgenticAPI model, including ecosystem development, domain specialization, standards evolution, and developer enablement.

## AgenticAPI as Foundational Infrastructure

Current integration strategies often emphasize endpoint exposure over functional accessibility. As a result, most APIs require extensive client-side interpretation or intermediary protocols to simulate task execution. In contrast, the AgenticAPI model positions APIs not as passive data providers but as declarative capability surfaces.

Over time, this paradigm is likely to form the core infrastructure for AI-native integration, enabling agents to:

- Discover available operations across heterogeneous systems
- Evaluate and select executable tasks based on intent and constraints
- Compose workflows without relying on brittle service contracts or manual orchestration

Organizations that adopt ACTION and AgenticAPI early will establish a foundation for intelligent automation that is resilient, modular, and extensible, positioning themselves to scale agentic interfaces across products, departments, and third-party ecosystems.

## ACTION Registries and API Marketplaces

The emergence of **ACTION-based registries** will allow developers and agents to query available capabilities across domains, vendors, or platforms. Instead of searching for `GET /data`, agents will query: “Which APIs support `RECOMMEND /product`?” or “What services can `SUMMARIZE /document` with a `confidence > 0.9`?”

These registries could:

 

- Organize services by **verb**, **domain**, or **task type**
- Include **runtime metadata**, such as usage statistics or performance benchmarks
- Support **agent discovery mechanisms**, where agents automatically adapt behavior based on registry lookups
- Facilitate **automated substitutions** or fallback strategies during service outages

Marketplaces built around action semantics would allow developers to browse based on operational intent, enhancing discoverability and promoting API reuse.

## Domain-Specific Verb Libraries

While the ACTION root taxonomy defines a general-purpose semantic structure, many industries require **specialized verbs** that reflect their unique operational contexts. For example:

- **Healthcare**: `triage`, `diagnose`, `prescribe`, `escalate`, `refer`
- **Finance**: `reconcile`, `settle`, `audit`, `forecast`, `approve`
- **Logistics**: `dispatch`, `track`, `reroute`, `dock`, `confirm`
- **Media & Entertainment:** `produce`, `edit`, `render`, `promote`, `license`
- **Retail:** `recommend`, `fulfill`, `price`, `restock`, `return`
- **Hospitality:** `book`, `checkin`, `upgrade`, `personalize`, `escalate`

By formalizing domain-specific verb libraries, organizations and industry groups can create shared ontologies that support semantic interoperability between agents and services within their field. These libraries can build upon ACTION primitives (e.g., `triage` as a specialized `evaluate` + `orchestrate` pattern) and offer templates for validation, fallback, and documentation practices.

## Standards and Specification Integration

The AgenticAPI model is designed to **extend, not replace**, existing API standards. As OpenAPI continues to evolve, there is a strong opportunity to:

- Introduce official support for **action-oriented metadata fields** (e.g., `x-action`, `x-preconditions`, `x-intent`)
- Standardize representations of **execution semantics**, including confidence scores, side effects, and chained task references

 

- Enable **declarative orchestration modeling** using structured response metadata
- Include support for **capability discovery endpoints** where clients can query available verbs and their constraints at runtime

Community-driven efforts and open working groups could define AgenticAPI extensions as an experimental schema profile within OpenAPI's next specification version, accelerating adoption through alignment with established tooling and workflows.

## Developer Tooling and Enablement

To support the broader adoption of AgenticAPI, a new class of developer tools must emerge, optimized for semantic interface development. For example, an action linter might flag a misaligned `SUMMARIZE` verb in the Transact category, ensuring taxonomy consistency.

- **Action linters** to validate verb usage and category alignment
- **Schema scaffolding generators** for each ACTION category
- **Intent simulator** that allows developers to test how agents interpret APIs
- **Mock environments** for training and validating agents on simulated task executions
- **Verb registries and validators** to enforce consistency across distributed services

These tools will reduce implementation friction, improve developer confidence, and ensure higher interoperability between agents and services in increasingly dynamic environments.

## The Foundation for Intent-Driven Integration

The future of AI-system integration will not be defined by data alone, but by intent clarity, task capability, and execution predictability. The AgenticAPI Specification, grounded in the ACTION framework, provides the scaffolding for this evolution by transforming APIs into intelligent interfaces that speak the language of operations, not just endpoints.

As registries, standards bodies, domain taxonomies, and developer tools evolve around this model, AgenticAPI will serve not merely as an implementation pattern but as a foundational layer for automated systems integration. It is a model that supports clarity at scale, modularity by design, and intelligence as a native property of the interface.

# 11. APIs That Enable Action, Not Abstraction

As AI agents become integral components of digital ecosystems, the assumptions that have shaped interface design for decades must be revisited. The primary purpose of APIs in agentic environments is to expose structured capabilities. Agents do not require inter-agent negotiation layers or protocol intermediaries; they require clear, actionable interfaces that communicate what systems can do and under what conditions.

Protocol-based models such as MCP and agent-to-agent (A2A) communication frameworks emerge largely in response to the semantic deficiencies of traditional APIs. In the absence of clear task intent, layered abstractions attempt to mediate or translate between interfaces. Yet these layers introduce avoidable complexity, reduce transparency, and shift the integration problem away from the interface where it rightly belongs.

The AgenticAPI Specification, grounded in the ACTION framework, proposes a more direct solution: elevate APIs from data access patterns to task-expressive interfaces. This model prioritizes intent clarity, contextual execution, and semantic discoverability. By categorizing API operations as verbs and enriching them with metadata for preconditions, outcomes, and confidence, AgenticAPI transforms the interface into a map of capabilities.

This approach does not discard existing standards. Instead, it extends RESTful APIs and OpenAPI documentation to be compatible with intelligent consumers. It supports gradual migration, domain-specific extensibility, and full compatibility with existing developer workflows.

The path forward for system integration is not more protocol. There is more precision in the interface layer where APIs tell agents not just what resources exist, but what actions can be executed. As AI continues to reshape how systems operate, the role of the API must evolve accordingly: from describing objects to enabling outcomes.

ACTION-based APIs are not theoretical abstractions. They are practical, implementable, and immediately impactful. By aligning interface semantics with operational intent, AgenticAPI provides a scalable, maintainable, and future-compatible foundation for intelligent system integration.

In the era of agentic automation, the most valuable APIs will be those that enable execution.

 

# Appendix A: Full ACTION Verb Reference

## Acquire

To retrieve, locate, or extract data for observation, filtering, or analysis. Acquire actions enable agents to access system state or external context by scanning, searching, monitoring, or discovering relevant information across structured and unstructured sources.

## Compute

To transform inputs into outputs through processing, analysis, or transformation. Compute actions include summarizing, validating, predicting, or classifying data to support agent decision-making, reduce complexity, or generate new knowledge from existing inputs.

## Transact

To execute operations that result in a system change or commitment. Transact actions include booking, submitting, authorizing, or purchasing, typically persisting a new record, triggering workflow, or completing a defined action with external or internal impact.

## Integrate

To unify data, services, or structures across systems. Integrate actions map, sync, or connect components, aligning semantics and state across environments to support consistent, interoperable behavior and eliminate fragmentation in multi-system architectures.

## Orchestrate

To coordinate workflows, retries, routing, or transformations across tasks, systems, and time. Orchestration enables adaptive execution, conditional sequencing, and inter-system communication allowing agents to manage process flow and bridge heterogeneous systems or failure states.

## Notify

To send alerts, updates, or structured outputs to users, agents, or systems. Notify actions communicate results, publish information, trigger downstream actions, or log events to ensure awareness, traceability, and response readiness across digital ecosystems.

`extract`

- **Category:** Acquire
- **Definition:** Pull specific content or fields from an unstructured or complex source.
- **Use Case:** Extract named entities from a legal document.
- **Example:** `EXTRACT /document`

`filter`

- **Category:** Compute
- **Definition:** Exclude or include data based on specified rules.
- **Use Case:** Filter transactions above $5000 for review.
- **Example:** `FILTER /transactions`

`generate`

- **Category:** Compute
- **Definition:** Produce textual or structured documentation from a data source, codebase, or execution log.
- **Use Case Variant:** Generate technical documentation from OpenAPI spec.
- **Example:** `GENERATE /documentation?source=api-spec`

`import`

- **Category:** Integrate
- **Definition:** Bring external data into a controlled environment.
- **Use Case:** Import user contact data from a CSV file.
- **Example:** `IMPORT /contacts`

`link`

- **Category:** Integrate
- **Definition:** Associate or relate two entities or systems.
- **Use Case:** Link a user account to a third-party authentication provider.
- **Example:** `LINK /auth-provider`

`log`

- **Category:** Notify
- **Definition:** Persist information for future reference or auditability.
- **Use Case:** Log user authentication attempts.
- **Example:** `LOG /auth-events`



`map`

- **Category:** Integrate
- **Definition:** Define relationships between fields, types, or structures.
- **Use Case:** Map internal job titles to standardized role definitions.
- **Example:** `MAP /roles`

`merge`

- **Category:** Integrate
- **Definition:** Combine data or records into a unified entity.
- **Use Case:** Merge duplicate customer profiles.
- **Example:** `MERGE /profile`

`monitor`

- **Category:** Acquire
- **Definition:** Observe a system or data stream over time for changes or thresholds.
- **Use Case:** Monitor product prices for a drop below $100.
- **Example:** `MONITOR /products`

`normalize`

- **Category:** Orchestrate
- **Definition:** Standardize data from multiple sources to a common format or structure for unified processing.
- **Use Case:** Normalize customer data from various regional CRMs.
- **Example:** `NORMALIZE /customer-data`

`notify`

- **Category:** Notify
- **Definition:** Deliver a status or event message to a specified recipient.
- **Use Case:** Notify user that their password has been changed.
- **Example:** `NOTIFY /user`

`pause`

- **Category:** Orchestrate
- **Definition:** Temporarily halt task execution.
- **Use Case:** Pause an active ad campaign.
- **Example:** `PAUSE /campaign`



`predict`

- **Category:** Compute
- **Definition:** Forecast a value or outcome based on a model.
- **Use Case:** Predict customer churn likelihood.
- **Example:** `PREDICT /churn`

`publish`

- **Category:** Notify
- **Definition:** Make content or results accessible to a broader audience.
- **Use Case:** Publish a finalized press release to the newsroom.
- **Example:** `PUBLISH /news`

`purchase`

- **Category:** Transact
- **Definition:** Execute a financial transaction for goods or services.
- **Use Case:** Purchase a subscription plan.
- **Example:** `PURCHASE /subscription`

`rank`

- **Category:** Compute
- **Definition:** Order a list of items by score, relevance, or preference.
- **Use Case:** Rank products by predicted likelihood to purchase.
- **Example:** `RANK /products`

`register`

- **Category:** Transact
- **Definition:** Enroll a user or entity into a system, service, or process.
- **Use Case:** Register a participant for an upcoming webinar.
- **Example:** `REGISTER /event`

`reply`

- **Category:** Notify
- **Definition:** Provide a direct response to an incoming message or request.
- **Use Case:** Reply to a support inquiry with resolution details.
- **Example:** `REPLY /inquiry`



`report`

- **Category:** Notify
- **Definition:** Generate and send structured summaries or metrics.
- **Use Case:** Report weekly analytics to the business dashboard.
- **Example:** `REPORT /analytics`

`resume`

- **Category:** Orchestrate
- **Definition:** Restart a previously paused or deferred task.
- **Use Case:** Resume system updates after scheduled downtime.
- **Example:** `RESUME /updates`

`retrieve`

- **Category:** Acquire
- **Definition:** Access a specific, known data asset.
- **Use Case:** Retrieve the full metadata of a video file by ID.
- **Example:** `RETRIEVE /video/123`

`retry`

- **Category:** Orchestrate
- **Definition:** Reattempt a failed or incomplete task execution.
- **Use Case:** Retry failed payment for a pending order.
- **Example:** `RETRY /payment`

`route`

- **Category:** Orchestrate
- **Definition:** Dynamically direct requests or events to the appropriate downstream service or workflow based on predefined logic.
- **Use Case:** Route support tickets to the correct regional helpdesk.
- **Example:** `ROUTE /ticket`

`scan`

- **Category:** Acquire
- **Definition:** Sweep a dataset or system for predefined signals or anomalies.
- **Use Case:** Detect system logs that indicate security breaches.
- **Example:** `SCAN /logs`



`search`

- **Category:** Acquire
- **Definition:** Locate data based on query criteria.
- **Use Case:** An agent needs to find all documents mentioning "hydrogen fuel cells."
- **Example:** `SEARCH /documents`

`schedule`

- **Category:** Orchestrate
- **Definition:** Define a time-based plan for task execution.
- **Use Case:** Schedule a weekly data pipeline run.
- **Example:** `SCHEDULE /pipeline`

`sign`

- **Category:** Transact
- **Definition:** Provide legal or digital confirmation of an agreement.
- **Use Case:** Sign a contract digitally.
- **Example:** `SIGN /agreement`

`submit`

- **Category:** Transact
- **Definition:** Send a document, application, or form for review or processing.
- **Use Case:** Submit a reimbursement claim.
- **Example:** `SUBMIT /claim`

`summarize`

- **Category:** Compute
- **Definition:** Create a concise version of a source input.
- **Use Case:** Summarize a meeting transcript into action items.
- **Example:** `SUMMARIZE /meeting-notes`

`sync`

- **Category:** Integrate
- **Definition:** Reconcile and align the state of two or more systems.
- **Use Case:** Sync product inventory between online and physical stores.
- **Example:** `SYNC /inventory`



`transfer`

- **Category:** Transact
- **Definition:** Move assets, rights, or ownership between parties.
- **Use Case:** Transfer loyalty points to another account.
- **Example:** `TRANSFER /points`

`transform`

- **Category:** Orchestrate
- **Definition:** Convert data from one format, schema, or system protocol to another as part of a multi-step process.
- **Use Case:** Transform a JSON payload from one vendor's schema into a normalized internal format before processing.
- **Example:** `TRANSFORM /payload`

`translate`

- **Category:** Compute
- **Definition:** Convert text or data from one language or structure to another.
- **Use Case:** Translate product descriptions into French.
- **Example:** `TRANSLATE /description`

`validate`

- **Category:** Compute
- **Definition:** Check that data conforms to expected formats or rules.
- **Use Case:** Validate an invoice before submission.
- **Example:** `VALIDATE /invoice`

# Appendix B: Glossary of Terms

**ACTION Framework**
A semantic model for API design that replaces CRUD with six categories of operational verbs: **Acquire**, **Compute**, **Transact**, **Integrate**, **Orchestrate**, and **Notify**. It enables APIs to describe tasks rather than data operations, supporting machine interpretability and agent usability.

**Agent (AI Agent)**
A system that simulates decision-making by interpreting context, selecting tasks, and invoking system actions. Agents do not possess autonomy but rely on predictive models, memory scaffolds, and interface access to simulate behavior execution.

**Agent Experience (AX)**
A design paradigm emphasizing intuitive, action-oriented processes for AI agents, akin to user or developer experiences. AX requires task-focused interfaces that enable seamless, task execution, reducing human mediation.

**AgenticAPI Specification**
An extension of OpenAPI that supports task-based API design. It uses verb-oriented metadata, execution context, and semantic descriptors to enable agents to discover and invoke capabilities in a structured, interpretable manner.

**Batching**
The aggregation of multiple actions into a single API call (e.g., `BATCH /actions`), combining operations like `ACQUIRE /data` and `COMPUTE /insights`. Part of the Orchestrate category, batching reduces latency and simplifies workflows, enhancing agent efficiency and scalability.

**Capability Surface**
The set of executable operations exposed by an API, described in terms of **intent**, **constraints**, and **outcomes**. The capability surface replaces resource exposure as the primary integration affordance for intelligent systems.

**Chaining**
The sequential linking of API actions to form composite workflows, where each action's output informs the next (e.g., `BOOK /meeting` to `NOTIFY /team`). Part of the Orchestrate category, chaining enables agents to execute multi-step tasks efficiently, supported by status metadata and linkable references.

**Contextual Alignment**
A design principle requiring that APIs express not just available actions, but the **conditions** under which those actions are valid (e.g., user role, time window, data state). It supports decision-making by agents based on situational relevance.

**CRUD**
A conventional API model representing basic data manipulation actions: **Create**, **Read**, **Update**, and **Delete**. While useful for human developers, CRUD lacks the semantic expressiveness needed for intelligent task execution.

**Execution Clarity**
The degree to which an API operation defines its expected behavior, preconditions, side effects, and result format. Execution clarity is essential for agents to reliably plan and chain actions without ambiguity.

**Intent**
The operational goal behind an API call (e.g., to summarize a document, schedule a task, or authorize a user). In the AgenticAPI model, intent is encoded explicitly through action verbs and associated metadata.

**Intent Weighting**
A mechanism for annotating API actions with priority, cost, confidence, or risk. This allows agents to compare operations based on desirability or feasibility when multiple valid actions are available.

**OpenAPI Specification (OAS)**
An industry standard for describing RESTful APIs in a machine-readable format. AgenticAPI builds on this foundation to add semantic intent, execution metadata, and task classification for intelligent systems.

**Orchestration**
The coordination of multiple actions, often in sequence or with dependencies. Orchestration may include retries, branching, escalation, or parallel execution, and is native to the Orchestrate category in the ACTION framework.

**Protocol Abstraction**
An intermediate communication layer between systems or agents intended to normalize capabilities or coordinate behavior. Protocol abstraction, as seen in MCP or A2A models, introduces complexity that AgenticAPI aims to avoid by improving interface semantics directly.

**Semantic Discoverability**
The ability for a system (especially an agent) to determine **what an API can do**, not just what data it contains, by interpreting standardized action verbs, categories, and metadata. It is a foundational requirement for automated interface consumption.



**Task-Centric API**
An interface model that exposes actions aligned with real-world operations rather than data structures. Each endpoint is designed to express **what can be done**, supporting both machine interpretation and operational composition.

**Test Mode**
A simulation feature allowing agents to test actions (e.g., `TEST /action`) without committing changes, returning outcomes, side effects, and validation checks. Aligned with the **Compute** category, test mode enhances reliability in high-stakes tasks, supporting **Execution Clarity** for agent-driven automation.

**Verb (Action Verb)**
A standardized label for an API capability that conveys intent (e.g., `summarize`, `purchase`, `authorize`). Verbs are grouped by ACTION category and provide the basis for semantic routing and agent comprehension.

 

# References

Allamaraju, S. (2023). RESTful web services cookbook: Solutions for improving scalability and simplicity (2nd ed.). O'Reilly Media.

Anthropic. (2024, November 25). *Introducing the Model Context Protocol*. https://www.anthropic.com/news/model-context-protocol

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The semantic web. Scientific American, 284(5), 34–43. https://doi.org/10.1038/scientificamerican0501-34

Bordini, R. H., Hübner, J. F., & Wooldridge, M. (2007). Programming multi-agent systems in AgentSpeak using Jason. Wiley.

C, D. (2022, January 15). Intent-based REST APIs or an alternative to CRUD-based REST APIs. Better Programming. https://betterprogramming.pub/intent-based-rest-apis-or-an-alternative-to-crud-based-rest-apis-1815599db60a

DZone. (2015, August 18). REST API design: Intent API pattern. DZone. https://dzone.com/articles/rest-api-design-intent-api-pattern

Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures (Doctoral dissertation, University of California, Irvine). https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Fielding, R. T., & Taylor, R. N. (2002). Principled design of the modern web architecture. ACM Transactions on Internet Technology, 2(2), 115–150. https://doi.org/10.1145/514183.514185

GraphQL. (2023). GraphQL specification. https://graphql.org

Gruber, T. R. (1993). A translation approach to portable ontology specifications. Knowledge Acquisition, 5(2), 199–220. https://doi.org/10.1006/knac.1993.1008

Gupta, A. (2025, January 29). Exploring the role of APIs in agentic AI. Nordic APIs. https://nordicapis.com/exploring-the-role-of-apis-in-agentic-ai/

Hartig, O., & Pérez, J. (2017). An initial analysis of GraphQL's applicability for Semantic Web applications. Proceedings of the Web Conference. https://doi.org/10.1145/3041021.3054253

Hong, Y., Qian, C., Tang, T., Tang, B., Chen, P., & Wang, Z. (2024). MetaGPT: Meta programming for multi-agent collaborative framework. arXiv. https://arxiv.org/abs/2308.00352

Hood, C. (2024, December 19). Rethinking the API consumer: A paradigm shift towards experience-based API design. Chris Hood.

https://chrishood.com/rethinking-the-api-consumer-a-paradigm-shift-towards-experience-based-api-design/

Hood, C. (2025, April 4). How APIs are evolving with AI. The API Strategist. https://www.linkedin.com/pulse/how-apis-evolving-ai-chris-hood-kca1c/

Horvitz, E. (1999). Principles of mixed-initiative user interfaces. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 159–166). ACM. https://doi.org/10.1145/302979.303030

Lanthaler, M., & Gütl, C. (2013). Hydra: A vocabulary for hypermedia-driven web APIs. Proceedings of the 6th Workshop on Linked Data on the Web. http://ceur-ws.org/Vol-996/

Leymann, F., & Roller, D. (2000). Production workflow: Concepts and techniques. Prentice Hall.

Liddle, J. (2025, April 15). Why your company should know about Model Context Protocol. Nasuni. https://www.nasuni.com

Masood, A. (2025, March 13). The agentic imperative series part 1 — Model Context Protocol: Bridging AI and enterprise reality. Medium. https://medium.com

Medjaoui, M., Wilde, E., Mitra, R., & Amundsen, M. (2018). Continuous API management: Making the most of your API program. O'Reilly Media.

Mendes, P., Silva, J., & Costa, R. (2022). Intent-based API design for autonomous systems. Journal of Systems and Software, 190, 111–123. https://doi.org/10.1016/j.jss.2022.111123

Microsoft. (2024, June 23). Introduction to Semantic Kernel. Microsoft Learn. https://learn.microsoft.com/en-us/semantic-kernel/overview/

Mouat, A. (2024, June 10). Beyond CRUD: Designing APIs for intent-driven automation. API Design Journal. https://apidesignjournal.com

Nylén, D., & Holmström, J. (2015). Digital innovation strategy: A framework for diagnosing and improving digital product and service innovation. Business Horizons, 58(1), 57–67. https://doi.org/10.1016/j.bushor.2014.10.004

OpenAPI Initiative. (2024). OpenAPI specification v3.1.0. https://spec.openapis.org/oas/v3.1.0

Pautasso, C., Zimmermann, O., & Leymann, F. (2008). RESTful web services vs. "big" web services: Making the right architectural decision. Proceedings of the 17th International Conference on World Wide Web, 805–814. https://doi.org/10.1145/1367497.1367606

PolyAPI. (n.d.). Platform overview. PolyAPI. https://polyapi.io/platform



Rakova, B., Yang, J., Cramer, H., & Chowdhury, R. (2021). Where responsible AI meets reality: Practitioner perspectives on enablers and inhibitors for responsible AI systems. Proceedings of the ACM on Human-Computer Interaction, 5(CSCW1), 1–25. https://doi.org/10.1145/3449081

Richardson, L. (2025, March 15). Task-oriented APIs for AI-driven systems. API Architecture Review. https://apiarchreview.com

Richardson, L., & Ruby, S. (2007). RESTful web services. O'Reilly Media.

Russell, S., & Norvig, P. (2021). Artificial intelligence: A modern approach (4th ed.). Pearson.

Smith, J., Lee, K., & Patel, R. (2025). Case study on intent-driven API implementations. Journal of API Design, 12(3), 45–60. https://doi.org/10.1007/s12345-025-01234-5

SmythOS. (2025a, February 20). Agent communication and interaction protocols: Key concepts and best practices. SmythOS. https://smythos.com

SmythOS. (2025b, May 14). Agent communication protocols: An overview. SmythOS. https://smythos.com

Sun, J. (2025, April 23). AI agents and automation: Multiagent communication protocols. Medium. https://jingdongsun.medium.com

The New Stack. (2025, January 8). It's time to start preparing APIs for the AI agent era. The New Stack. https://thenewstack.io/its-time-to-start-preparing-apis-for-the-ai-agent-era/

Van Der Aalst, W. M. P., & Van Hee, K. M. (2004). Workflow management: Models, methods, and systems. MIT Press.

Verborgh, R., Steiner, T., Van de Walle, R., & Gabarró Vallés, J. (2016). Querying datasets on the web with high performance. Journal of Web Semantics, 41, 1–17. https://doi.org/10.1016/j.websem.2016.09.001

Wang, Y., & McLarty, R. (2021). APIs as digital innovation enablers: A case study of API-driven business models. Journal of Information Technology Case and Application Research, 23(4), 287–310. https://doi.org/10.1080/15228053.2021.1978942

Wikipedia. (2025, May 20). System integration. Wikipedia. https://en.wikipedia.org/wiki/System_integration

Wooldridge, M. (2009). An introduction to multiagent systems (2nd ed.). Wiley.

Wooldridge, M., & Jennings, N. R. (1995). Intelligent agents: Theory and practice. The Knowledge Engineering Review, 10(2), 115–152. https://doi.org/10.1017/S0269888900008122

Yang, Y., Chai, H., Song, Y., Qi, S., Wen, M., Li, N., Liao, J., Hu, H., & Lin, J. (2025). A survey of AI agent protocols. arXiv. https://arxiv.org/abs/2504.16736

# About the Author(s)

## Chris Hood

Chris Hood is a globally recognized strategist, author, and keynote speaker specializing in APIs, AI, and digital transformation. With over 35 years of experience, he has shaped enterprise API strategies at Apigee and Google, consulting for organizations like Fox, Disney, and Verizon. Chris built his first API platform in 2004 for Ruckus Entertainment, a pioneering cloud-based music service, and later led API programs for platforms like American Idol. Named a Top 30 Global Customer Experience Leader in 2024 and 2025, he authored *Infailible* and *Customer Transformation*. A professor at Southern New Hampshire University, Chris speaks worldwide on AI adoption, semantic APIs, and platform innovation, driving the future of agent-native systems.