Capability-Based Model Gateway Architecture for Multi-Agent Fleets

Executive summary

The proliferation of advanced language models, specialised reasoning engines, and varied inference platforms necessitates a structural paradigm shift in how autonomous AI agents are orchestrated. Tightly coupling an agent's definition to a specific provider's physical model name, proprietary API format, and idiosyncratic prompt structure inevitably leads to vendor lock-in. This architectural fragility manifests catastrophically during unexpected provider outages, model deprecations, or sudden pricing adjustments. Conversely, attempting to abstract providers via a lowest-common-denominator interface strips away the advanced capabilities—such as continuous reasoning logic, strict context-free grammar structured outputs, and prompt prefix caching—that make modern inference engines powerful and economically viable.

The optimal architectural solution lies in designing a capability-based model gateway. In this paradigm, a portable agent fleet requests capabilities, operational constraints, and service-level objectives rather than explicitly hard-coding physical model strings. A deterministic, highly available model gateway dynamically resolves these abstract requests into approved providers, physical models, deployments, provider-specific prompts, optimal reasoning settings, token budgets, and fallback chains. This exhaustive investigation establishes that a gateway architecture must expose a stable, universal common core while preserving optional, provider-specific enhancements through explicit and versioned extensions.

Key conclusions

Extensive technical analysis validates the core hypothesis: portable agent fleets must operate independently of physical model constraints through capability negotiation. The primary conclusions drawn from evaluating production-grade model routing architectures include the following critical insights.

First, capability routing inherently outperforms name-based routing. Mapping requests to a dynamic capability registry prevents workflow failure during model deprecation and enables immediate exploitation of newly released inference engines without modifying downstream agent logic. Second, strict adapter segregation is mandatory. Transport adapters managing HTTP protocols, provider adapters handling authentication, model adapters dictating prompt logic, and capability adapters emulating missing features must remain strictly segregated to prevent architectural bleed. Third, governance and privacy must act as hard routing constraints. Data privacy frameworks, such as enforcing European Union data boundaries for personally identifiable information, must function as deterministic disqualifying filters in the routing layer rather than relying on probabilistic instructions within a system prompt.[1]

Furthermore, architectural emulation cannot fully replace native support. While a gateway can

emulate missing capabilities—such as prompting for JSON when strict constrained decoding is natively unavailable—the system must explicitly track whether a capability is native or emulated to accurately calculate reliability confidence.[3] Finally, multi-turn state management has become highly provider-specific. Advanced reasoning features, such as preserving internal encrypted reasoning blocks during tool-use loops, demand gateway-level state management to shield agents from provider-specific multi-turn rules.[4]

Terminology and conceptual model

To establish a rigorous architectural foundation, it is necessary to strictly separate logical abstractions from physical infrastructure. The following taxonomy defines the conceptual model of the capability-based architecture.

Term	Definition and Architectural Position	Provider Independence	Example and Misunderstandings
Model	The foundational neural network weights.	Provider-specific.	E.g., Claude 5. Misunderstood as interchangeable with "deployment."
Model family	A lineage of models sharing prompt formats and systemic behaviours.	Provider-specific.	E.g., OpenAI GPT-5.5 family.
Model version	A specific, immutable training snapshot.	Provider-specific.	E.g., gpt-4o-2024-08-06.
Model provider	The corporate entity creating the model weights.	Independent concept, specific instance.	E.g., Anthropic, DeepSeek.
Inference provider	The entity hosting the model.	Independent concept.	E.g., Microsoft Azure, AWS Bedrock.

Inference endpoint	The specific network address serving requests.	Provider-specific.	E.g., https://api.openai.com/v1/chat.
Deployment	A specific instantiation of a model at an endpoint, governed by regional hardware limits.	Provider-specific.	E.g., Azure West Europe standard deployment.
Local model	Weights executed on owned infrastructure.	Independent concept.	E.g., Llama 3 running on local vLLM.
Managed model	Weights executed on managed, proprietary infrastructure.	Independent concept.	E.g., Google Gemini Pro.
Open-weight model	Models with publicly accessible weights.	Independent concept.	E.g., DeepSeek V4.
Proprietary model	Models with closed, inaccessible weights.	Independent concept.	E.g., Claude Opus 4.8.
Model adapter	Translates universal prompt structures into model-specific formats.	Provider-specific implementation.	E.g., System prompt formatting adapter.
Provider adapter	Normalises authentication, headers, and global error mapping.	Provider-specific implementation.	E.g., Translating Azure Entra ID tokens.
Transport adapter	Normalises network	Independent	E.g., SSE to WebSockets

	protocols.	implementation.	bridge.
Tool adapter	Normalises execution schemas and tool-result formats.	Provider-specific implementation.	E.g., Converting JSON schema to strict formats.
Capability adapter	Emulates a logical capability when a model lacks native support.	Provider-specific implementation.	E.g., Emulating JSON mode via prompting.
Model gateway	Centralised proxy and control plane mediating agent traffic.	Independent architecture.	E.g., LiteLLM proxy router.[6]
Capability registry	Control-plane database tracking model features.	Independent architecture.	Tracks which models natively support vision.
Logical model	An abstract alias mapped dynamically to physical resources.	Independent concept.	E.g., coding.high. Misunderstood as an API key.
Physical model	The actual string accepted by an inference API.	Provider-specific.	E.g., claude-3-opus-20240229.
Reasoning effort	Normalised request translated into specific compute controls.	Independent concept.	E.g., "high", mapping to Anthropic's xhigh effort.[4]
Structured output	Output generated using constrained decoding.	Independent concept.	E.g., OpenAI strict: true JSON schemas.[3]

Feature detection	Automated verification of a model's true capabilities.	Independent architecture.	Probing to see if a model supports parallel tools.
Circuit breaker	Automatic traffic cessation upon consecutive failures.	Independent architecture.	Halting routing to an endpoint after 5 timeout errors.[7]
Fallback	Sequential routing to an alternate deployment upon primary failure.	Independent architecture.	Rerouting from Azure OpenAI to AWS Bedrock.[8]
Evaluation gate	The staging process before a model accepts live traffic.	Independent architecture.	Canary testing a new open-weight model.

Agent definition versus model definition

An architecture inevitably fails when agent configurations merge with physical model constraints. An agent definition must solely outline the logical entity: its role, purpose, required tools, data-access permissions, workflow constraints, memory policy, and required capabilities (such as "high-quality reasoning" or "vision processing"). A model definition dictates physical characteristics: context limits, supported modalities, explicit tool-call parameter structures, token costs, regional availability, and historical reliability.

Merging these definitions binds the agent's logic to a specific snapshot in time. If a provider renames a model, deprecates an endpoint, or alters pricing, the agent's source code breaks. By separating them, the same named agent can execute a coding task via a premium cloud reasoning engine during peak operations, shift to a local open-weight model for confidential repository analysis, and gracefully degrade to a cheaper cloud model during an upstream outage. This substitution occurs without altering the agent's system instructions or underlying workflow logic. Acceptable behavioural changes during substitution include variations in latency or response verbosity. Unacceptable changes include violating strict schema contracts, executing hallucinated tools, or disregarding data residency rules.

Provider, model, capability, and transport adapters

To achieve model independence without sacrificing advanced features, the gateway must implement distinct, layered adapters. Blurring these lines leads to fragile, unmaintainable

monolithic codebases.
The transport adapter is exclusively responsible for network communication, retry logic, connection pooling, and protocol negotiation. It handles establishing HTTP Keep-Alive connections and translating Server-Sent Events (SSE) into asynchronous generator streams, completely agnostic of the payload's semantic meaning.
The provider adapter sits above the transport layer. It normalises authentication, rate-limit header interpretation, and provider error mapping. Its responsibility is to inject Authorization: Bearer headers and map provider-specific error codes (such as an Anthropic overload error or an OpenAI 429 Too Many Requests) into a canonical internal error state.[7]
The model adapter handles behavioural and feature differences for a particular model family. Its core responsibility includes prompt packing, system prompt formatting, and token overhead calculation. If a legacy model family requires system instructions to be wrapped in specific XML tags while a newer family requires Markdown headers, the model adapter executes this transformation transparently.
The capability adapter converts a logical capability into a model-specific implementation, often relying on emulation. If an agent requires structured output, but the routed model lacks native constrained decoding, the capability adapter dynamically alters the system prompt to enforce JSON structure, parses the raw string response, and executes internal validation and repair loops.
These adapters must remain separate. A single adapter is only sufficient in simplistic, experimental frameworks; production systems must be able to hot-swap a transport adapter (for instance, moving to a gRPC backend) without rewriting prompt-formatting logic.

Unified model interface

A provider-neutral model invocation interface must encapsulate all potential request requirements while isolating provider-specific parameters into explicitly versioned extensions. This prevents the primary schema from devolving into a bloated configuration dump.
Mandatory request fields must include a unique request_id, the canonical messages array, an explicit capability_request object detailing required and preferred traits, and maximum budget thresholds for both cost and latency. Standard optional fields include canonical tools, required_output_schema, a normalized reasoning_budget, and privacy_classification.
To avoid the lowest-common-denominator failure, the interface must accept a provider_extensions object. This structure (e.g., provider_extensions: { "anthropic": { "cache_control": "ephemeral" } }) allows the agent orchestrator to suggest provider-specific optimisations. The gateway passes these extensions through only if the routed provider matches the extension key; otherwise, they are safely ignored.
The canonical response interface standardises the output. It returns the selected_provider, selected_model, and physical deployment. It includes the content payload, normalised tool_calls, and strictly validated structured_output. Crucially, it returns exhaustive telemetry encompassing estimated and actual costs, latency, retry history, and fallback paths executed during the request lifecycle.

Capability taxonomy

To route tasks effectively, the system must categorise requests using a highly granular capability taxonomy rather than model names.
Reasoning capabilities denote the cognitive complexity required, ranging from simple classification and ambiguity resolution to complex multi-step planning and mathematical reasoning. Coding capabilities specify the depth of software engineering required, distinguishing between basic code generation, repository-scale reasoning, secure coding analysis, and autonomous terminal tool use.
Tool-use capabilities differentiate between basic function calling, sequential tool loops, parallel tool execution, and advanced native computer use. Data capabilities encompass structured extraction, schema mapping, tabular reasoning, and document parsing. Multimodal capabilities specify requirements for image understanding, audio transcription, and screenshot interpretation.
Finally, operational characteristics define the non-functional requirements of the request. These include strict low latency, high throughput, zero-data-retention requirements, private local deployment mandates, prompt caching support, and deterministic decoding availability. An agent constructs a structured request specifying hard requirements (which disqualify non-compliant models) and soft preferences (which serve as optimisation weights).

Capability registry

The capability registry functions as the authoritative control-plane database, containing both declared and verified model characteristics. Provider documentation is frequently aspirational or temporally inaccurate; therefore, the registry must stratify capabilities based on their verification source.
A provider-declared capability is extracted from official API documentation. An adapter-detected capability is verified via automated startup probes (for instance, submitting a dummy parallel tool request to verify true support). A benchmarked capability is verified through static test suites run during the model onboarding phase. A production-observed capability is continuously updated via live telemetry, tracking metrics such as schema-validity rates and tool-call success rates over rolling time windows.
Relying solely on provider documentation dictates routing failure. The registry employs a confidence and freshness model. If a deployment has not processed a specific capability request within a configured time horizon, its confidence score decays. This decay forces the routing engine to deprioritize the deployment for high-stakes tasks until background evaluation probes refresh the confidence score.

Capability-based routing

Model selection relies on sophisticated, hybrid routing algorithms evaluated continuously against live capability registry metrics.
Rules-based routing evaluates hard constraints deterministically. A request tagged with a strict privacy classification immediately filters out shared global endpoints, restricting candidates to

local models or sovereign cloud deployments. Weighted scoring is subsequently applied to the eligible pool. Dimensions such as historical quality, cost, latency, and cache hit probability are scored against the agent's soft preferences.

Constraint optimisation algorithms select a model that satisfies all hard constraints while optimising for a primary objective, such as minimising cost while ensuring latency remains under a specific percentile. Historical-performance routing utilizes telemetry to favour models that have demonstrated high empirical success rates for similar task types in the recent past.

Health-check driven routing integrates tightly with circuit breakers; if active background probes or live traffic detect elevated error rates, the routing algorithm immediately depreciates the deployment's weight to zero, removing it from the candidate pool before user traffic is impacted.[7]

Avoiding model-name routing

Embedding physical model names directly into agent prompts, workflows, task schemas, or database records constitutes a critical anti-pattern. Logic such as "Always use gpt-4o for coding" or "Send research tasks to claude-3-5-sonnet" is immensely fragile.

Such hard-coded logic breaks instantly when a provider renames a model, deprecates a version, alters pricing structures exponentially, or experiences a regional outage. It also prevents the agent from seamlessly upgrading to a superior, newly released model.

The architecture resolves this by employing logical aliases and abstract capability requests. Logical aliases such as coding.high, reasoning.deep, classification.bulk, and local.private serve as stable routing keys. The gateway resolves these aliases dynamically at runtime by querying the capability registry, ensuring that the underlying physical infrastructure can evolve, degrade, or scale without requiring a single modification to the agent's source code.

Tool-call normalisation

The fragmentation of tool-calling schemas represents a primary vector for vendor lock-in. Providers enforce radically different constraints regarding JSON Schema versions, parameter handling, enum validation, nullable fields, and parallel tool invocation.[10]

The gateway must implement a canonical internal tool schema based on standard JSON Schema definitions. The tool adapter translates this canonical schema into provider-specific tool definitions. When handling unsupported schema features, the adapter employs deterministic strategies. For strict providers, it may apply schema simplification or deterministic rejection if simplification would alter the tool's execution safety.

Furthermore, the adapter normalises tool_call_id variables and ensures that tool_result messages accurately map back to the original identifiers regardless of the provider's native formatting requirements. Crucially, tool execution authorisation remains entirely outside the model adapter. The gateway merely normalises the model's intent to call a tool, hands the canonical payload back to the agent orchestrator for secure execution, and awaits the result.

Structured-output compatibility

Ensuring reliable, strictly typed JSON output exposes significant architectural divergence between providers. Relying merely on "JSON mode" is wholly insufficient, as it only guarantees syntactical validity, not schema adherence or type correctness; fields marked as required may still be silently omitted.[3]

OpenAI's strict structured outputs utilize context-free grammar constraints to guarantee compliance at the decoding level, physically preventing the generation of invalid tokens.[3] However, this strict mode prohibits standard JSON schema features: it explicitly rejects default values, requires additionalProperties: false on every nested object, and demands that all properties be listed in the required array, forcing developers to use nullable union types for optional fields.[3]

A provider-independent structured-output layer must abstract these limitations. The capability adapter accepts a standard schema and dynamically mutates it—stripping defaults and enforcing strict bounds—before submitting it to strict-compliant providers. For models lacking native constrained decoding, the adapter injects the schema into the system prompt, parses the resulting text, and executes automated retry-and-repair loops, surfacing a deterministic failure to the agent orchestrator if the contract cannot be ultimately fulfilled.

Prompt portability

A singular, static prompt rarely performs optimally across varied model families due to inherent differences in pre-training and alignment tuning. Models exhibit varying sensitivities to XML tags, Markdown formatting, instruction hierarchies, few-shot example placement, and negative constraints.

Prompt portability necessitates a layered composition architecture. The core instructions—defining the agent's role, primary task, and general constraints—remain strictly provider-neutral. The gateway introduces a model-family optimisation layer that intercepts this baseline prompt prior to execution. This layer restructures the payload, perhaps moving tool instructions into specific XML blocks for Claude models or shifting negative constraints to the end of the user prompt for specific open-weight models. This architectural isolation prevents provider-specific prompt engineering from permanently contaminating the core agent definitions.

Provider-specific prompt optimisation

To prevent the accumulation of uncontrolled, spaghetti-code prompt patches across the codebase, prompt composition must follow a strict, deterministic hierarchy.

The composition process begins with the global policy layer, enforcing non-negotiable organisational rules such as data privacy constraints. It merges with the agent definition, workflow constraints, and task instructions. The generic tool protocol is appended. Subsequently, the model-family optimisation layer applies structural reordering. Finally, a provider-specific shim injects minor API-required text artifacts.

The initial layers are authoritative and authored by developers. The latter optimisation layers are managed exclusively by versioned capability adapters within the gateway. This separation ensures that an agent remains portable while still aggressively exploiting the native

instruction-following idiosyncrasies of a specific model family.

Context-window abstraction

Advertised context windows rarely reflect effective, safely usable context limits. Furthermore, multi-modal token consumption, hidden reasoning token overhead, and extensive tool-schema parameter definitions rapidly consume token budgets before any actual generation begins. Assuming a task fits merely because the advertised limit is large guarantees production runtime errors.

The gateway incorporates a context-capacity service that estimates token usage prior to invocation, utilizing provider-specific tokenisers where available. It reserves necessary output capacity and tool-call capacity, explicitly accounting for hidden reasoning tokens, and applies a strict safety margin. If an agent's request exceeds the effective limit, the router triggers graceful degradation strategies. These may include hierarchical retrieval, map-reduce document chunking, or automated escalation to a model with a verified, massive context window, such as DeepSeek V4 Pro's 1M-token context capability.[12]

Reasoning and token-budget abstraction

Advanced models increasingly utilize inference-time compute logic, where the model generates hidden reasoning tokens prior to outputting visible text. Providers implement and control this mechanism radically differently. Anthropic has actively deprecated manual budget_tokens for newer models, shifting to an adaptive thinking mode controlled by an output_config: {"effort": "medium"} parameter that dictates reasoning intensity.[4] OpenAI similarly utilises a reasoning_effort parameter (low, medium, high) for its o-series reasoning models.[14]

The gateway abstracts these disparate mechanisms via a normalised request parameter, such as reasoning_budget: "high". The capability adapter dynamically translates this into the provider's specific API requirements. Critically, during multi-turn tool use, providers like Anthropic require the API caller to pass the generated thinking blocks—including encrypted signature fields—verbatim back into the conversation history to preserve reasoning continuity.[5] The gateway must manage this complex multi-turn state invisibly, shielding the orchestrating agent from provider-specific message retention rules.

Streaming normalisation

Streaming implementations vary drastically across Server-Sent Events (SSE), WebSockets, and proprietary chunk structures. Providers differ significantly in how they emit tool-call deltas, incremental reasoning tokens, usage statistics, and connection interruptions.

The gateway exposes a canonical internal event stream, translating disparate protocols into a unified, predictable sequence (e.g., response.started, content.delta, tool_call.arguments_delta, usage.updated, response.completed). If a requested provider does not support streaming, the adapter waits for the synchronous response and simulates the event stream. This ensures the agent's downstream consuming logic requires only a single, simplified code path regardless of

the upstream implementation.

Error normalisation

Raw provider errors, such as a generic HTTP 400 or 503, provide insufficient contextual telemetry for autonomous workflows to make recovery decisions. The gateway translates upstream errors into a standard internal taxonomy.

Following advanced proxy implementations, the gateway maps errors to determine whether the failure originated from an upstream API failure (which is retryable) or an internal gateway misconfiguration (which is non-retryable).[18] A 429 Too Many Requests translates to a RateLimitError prompting jittered exponential backoff. A 400 Bad Request regarding a malformed structured output schema translates to a DeterministicRequestError, which immediately halts retries and fails back to the agent orchestrator for workflow correction, preventing futile retry loops that waste both time and capital.

Retry and fallback design

Robust multi-agent fleets rely heavily on bounded retries and capability-aware fallback chains. The system differentiates fundamentally between network failures (which trigger a retry on the same deployment) and provider overload or region unavailability (which trigger a retry on an alternate deployment).

Fallback chains must be constructed dynamically based on capabilities, not arbitrary, hard-coded model names.[7] If a primary high-tier reasoning model fails, the fallback router searches the capability registry for another deployment that strictly satisfies the agent's baseline constraints. Crucially, sequential fallbacks are treated as entirely fresh requests; semantic caching checks, telemetry recording, and governance validation must be re-run for the fallback provider to ensure behavioural consistency.[7] Explicit maximum-attempt budgets prevent retry storms and exponential cost growth.

Feature detection

Because provider capabilities mutate silently and marketing documentation frequently precedes actual API availability, relying on static configuration is inadequate. The gateway employs active feature detection via continuous production telemetry and targeted synthetic feature tests.

When a new deployment is registered, it undergoes automated probing to confirm genuine support for structured output strictness, parallel tool calling, and maximum effective context limits. If a synthetic probe fails, the specific capability is marked as unsupported in the registry. This prevents the router from assigning incompatible workloads to that deployment, ensuring that feature claims are independently verified before exposing the deployment to live agent traffic.

Model evaluation and onboarding

A rigorous evaluation gate is required to prevent systemic regressions when introducing new

models to the fleet. Before a new model accepts broad production traffic, it progresses through tightly controlled validation stages.
The process begins with registration and automated probing to detect declared capabilities. It advances to synthetic benchmarking to validate tool-call reliability and strict instruction following. If successful, the gateway routes a percentage of live shadow traffic to the new model, measuring latency and cost without returning the result to the user. Following positive telemetry, a limited canary deployment allows the model to serve a small fraction of live agent decisions, closely monitored for anomalous failure rates or safety refusals. Only after passing these empirical gates does the model enter the general availability routing pool. Generic public benchmarks are insufficient; routing decisions must rely on task-specific evaluation results derived from the organisation's actual agent fleet traffic.

Model-specific optimisation without lock-in

To avoid lowest-common-denominator interfaces, the architecture employs progressive enhancement. The core baseline path remains universally portable. However, agents can specify optional capabilities, such as provider-native prompt caching or specific inference-time compute controls.
The gateway evaluates if the selected physical model supports the requested enhancement. If it does, the adapter natively executes the feature. If it does not, the gateway either emulates the feature (such as utilizing a standard Python execution tool if native code execution is absent) or ignores it if deemed non-critical. This architectural pattern allows fleets to aggressively exploit cutting-edge provider capabilities while retaining the structural ability to route around outages using baseline fallback models.

Avoiding lowest-common-denominator design

A simplistic abstraction layer inevitably strips messages of multimodal input, limits tasks to single sequential tool calls, and ignores advanced reasoning controls merely to maintain a uniform API surface.
To combat this, the gateway implements a tiered interface. The core portable interface guarantees features expected across all providers. Standard optional capabilities expose common but not universal features, like reasoning effort scaling. Provider extensions allow explicit, opt-in configurations (e.g., specific caching headers) stored in a segregated payload object. Experimental capabilities are gated behind evaluation flags. Agents and workflows declare which capability tiers they require, and the router strictly enforces that advanced requests are matched only to capable deployments, ensuring advanced features are leveraged rather than suppressed.

Local and cloud hybrid architecture

A robust platform seamlessly orchestrates both local endpoints (e.g., local vLLM instances, desktop runtimes) and proprietary cloud APIs. Local models possess distinct operational profiles: they offer absolute data privacy, zero marginal per-token cost (offset by underlying hardware and infrastructure electricity costs), and offline availability. However, they frequently

exhibit lower concurrency, restricted context limits, and inferior tool-calling reliability compared to flagship proprietary models.
Routing rules dynamically enforce data governance across this hybrid fleet. Tasks classified as highly confidential or containing personally identifiable information are strictly routed to local models. High-volume, low-complexity classification tasks are routed to local instances to minimise API expenditure, while complex, ambiguity-heavy strategic synthesis tasks are escalated to approved cloud reasoning models. The unified registry maintains distinct capability profiles for local models, ensuring the router understands their specific operational constraints.

Open-source model support

Integrating open-weight models requires acknowledging that an "OpenAI-compatible API" endpoint does not guarantee behavioural compatibility. Adapters must account for diverse chat templates, varying tokeniser efficiencies, and differing tool-use fine-tuning quirks that cause models to hallucinate arguments or ignore schema constraints.
The gateway's model adapters adjust prompts specifically for models running on high-throughput inference servers, normalising syntax and managing speculative decoding latency variations invisibly to the agent. Open-weight models, such as DeepSeek V4, offer massive context windows and aggressive prompt caching economics [12], requiring the router to evaluate open-source models based on their unique performance profiles rather than treating them as generic OpenAI clones.

Gateway architecture comparison

The physical deployment pattern of the gateway significantly affects latency, scalability, and security isolation.

- **Embedded library**: A thin unified client library is simplest for a solo builder but lacks centralised policy control, making credential rotation and global rate-limiting impossible.
- **Sidecar gateway**: Running a proxy alongside the agent runtime provides excellent isolation but duplicates cache and rate-limit states across nodes.
- **Central shared gateway**: Ideal for multi-fleet observability and global rate limiting, but introduces a potential single point of failure.
- **Hierarchical/Federated gateway**: Local fleets execute fast, low-latency routing via edge gateways while asynchronously synchronising usage ledgers and global policies with a central control plane.

For advanced builders and enterprise fleets, a federated approach with a highly available central gateway provides the optimal balance of governance and resilience.

Privacy, regional routing, and credentials

Data residency is a non-negotiable routing constraint. For instance, strict compliance with the European Union Data Boundary requires explicit deployment settings. Microsoft Azure OpenAI processing requires the selection of the "Data Zone Standard" or "EU Sovereign Cloud"

deployment types to guarantee that input, output, and system telemetry remain strictly within the EU; the default "Worldwide Standard" deployment type violates strict GDPR residency rules by probabilistically routing data globally for load balancing.[1]

Privacy constraints must operate as hard routing filters. A task tagged with a European privacy classification automatically disqualifies any deployment lacking verified regional isolation. Furthermore, a secret-broker architecture ensures that agents never handle raw provider API keys. The gateway injects provider credentials just-in-time during the transport adapter phase, selecting the correct billing account based on the originating tenant, project, and selected region.

Caching and batch processing

Prompt caching significantly alters cost optimization routing algorithms. For example, DeepSeek V4 Flash advertises $0.14 per million input tokens, but a cache hit reduces this to $0.0028 per million tokens—a 98% reduction.[1][2] The router must track prefix cache probabilities; a model with a higher base cost but an established cache for a specific massive system prompt may be exponentially cheaper overall than a model with a lower base cost but no cache history.

For latency-insensitive tasks, the gateway exposes an asynchronous batch interface. It accumulates requests and dispatches them to provider-native batch APIs (which typically offer 50% pricing discounts).[4] Workflows dynamically choose between synchronous streaming calls for user-facing actions and batch processing for background classification.

Observability and production feedback

Provider-neutral model telemetry must capture logical metrics to evaluate performance accurately. The observability pipeline records the logical capability requested, routing reason, fallback reason, and execution duration. It strictly distinguishes between provider-reported usage, adapter-estimated usage (used for budget enforcement before the request completes), and gateway-calculated cost.

This telemetry feeds continuous quality and reliability feedback loops. If an agent frequently retries a specific model due to schema failures, the gateway's feedback service automatically degrades that model's capability score for that specific task type. This contextual bandit approach naturally routes future traffic away from degrading models and toward more reliable alternatives, preventing sudden model overexposure while avoiding the trap of comparing models across divergent task distributions.

Multi-fleet governance

A global gateway enforces organizational policy across disparate, specialised agent fleets. A coding fleet may be permitted broad access to high-cost reasoning models for repository analysis, while a customer-support fleet is constrained strictly to high-throughput, low-latency models. A compliance fleet may be hard-locked to local, zero-data-retention deployments.

The control plane manages these varying cost ceilings, privacy rules, and capability profiles centrally. This allows a global model platform to enforce strict security and financial governance

while providing fleet orchestrators the autonomy to dynamically route tasks within their approved boundaries.

Proposed Capability-Based Model Gateway

The reference architecture relies on the strict separation of the control plane and the request plane. The agent interacts solely with the unified contract.

Key Components:

1. **Unified Model API**: The single ingress point enforcing the Agent-to-gateway contract.
2. **Policy Engine**: Evaluates strict budgets, privacy routing, and regional restrictions.
3. **Capability Router**: Matches the request to the Capability Registry and Deployment Registry.
4. **Adapters**: Transport, Provider, Model-family, and Tool-call normalisers translate the canonical request.
5. **Context-Budget Manager**: Calculates effective limits and token/cost estimates prior to execution.
6. **Reliability Managers**: Retry manager, fallback manager, and circuit-breaker service ensure failure isolation.
7. **Telemetry & Feedback**: Usage ledger, observability dashboard, and production-feedback service update historical routing scores based on success rates.

Architecture diagrams

Code snippet

```
flowchart TD
  subgraph Agent Fleets
    A1[Coding Agent]
    A2
  end

  subgraph Unified Model Gateway
    API[Unified API Contract]
    PR
    CR

    subgraph Registries
      CapReg
      DepReg
    end

    TA
```

```
    MA[Model Adapters]
    PA[Provider Adapters]
  end

  subgraph Inference Infrastructure
    C1[Cloud Provider A]
    C2
    L1[Local vLLM Cluster]
  end

  A1 --> API
  A2 --> API
  API --> PR
  PR --> CR
  CR <--> CapReg
  CR <--> DepReg
  CR --> TA
  TA --> MA
  MA --> PA
  PA --> C1
  PA --> C2
  PA --> L1
```

Code snippet

```
sequenceDiagram
  participant Agent
  participant Gateway
  participant Registry
  participant Provider

  Agent->>Gateway: Request (Capabilities, Schema, Budget)
  Gateway->>Gateway: Validate Policy & Privacy
  Gateway->>Registry: Fetch Eligible Deployments
  Registry-->>Gateway: List of Candidates
  Gateway->>Gateway: Score & Select Deployment
  Gateway->>Gateway: Translate Tools & Prompt
  Gateway->>Provider: Execute HTTP Request
  Provider-->>Gateway: Raw Stream / Response
  Gateway->>Gateway: Normalise Output & Validate Schema
  Gateway-->>Agent: Canonical Response
```

```
    Gateway->>Registry: Update Usage & Health Metrics
```

```
Code snippet
flowchart LR
    A --> B{Tool Normaliser}
    B -->|OpenAI Adapter| C
    B -->|Anthropic Adapter| D[input_schema format, xml wrappers]
    B -->|Local Adapter| E
```

```
Code snippet
stateDiagram-v2
    [*] --> PrimaryDeployment
    PrimaryDeployment --> Success : 200 OK
    PrimaryDeployment --> ProviderFallback : 429 / 503 Timeout
    PrimaryDeployment --> ModelFallback : 400 Bad Schema
    ProviderFallback --> Success : 200 OK
    ProviderFallback --> LocalFallback : Network Outage
    LocalFallback --> Success : 200 OK
    LocalFallback --> HumanEscalation : Local Failure
    Success --> [*]
    HumanEscalation --> [*]
```

```
Code snippet
flowchart TD
    A --> B
    B --> C
    C --> D
    D --> E
    E[Model Family Optimisation Layer] --> F
    F --> G
```

```
Code snippet
classDiagram
    class ModelProfile {
        +String logical_id
        +String provider
        +List modalities
    }
    class CapabilityRecord {
        +String capability_name
        +Boolean declared
        +Boolean detected
        +Float production_success_rate
        +DateTime last_verified
    }
    ModelProfile "1" *-- "many" CapabilityRecord : possesses
```

```
Code snippet
flowchart TD
    Req
    Req --> Check{Data Sensitivity?}
    Check -->|Confidential / PII| Local
    Check -->|Public Data| Cloud
    Local --> Evaluate
    Cloud --> Evaluate
    Evaluate -->|Simple| Cheap[Low-Cost Models]
    Evaluate -->|Complex| Expensive
```

```
Code snippet
flowchart TD
    LB
    LB --> RP1
    LB --> RP2
    RP1 --> Redis
    RP2 --> Redis
    RP1 --> CP[Control Plane]
    RP2 --> CP
    CP --> DB
```

```
    RP1 --> CB1{Circuit Breaker}
    RP2 --> CB2{Circuit Breaker}
    CB1 --> Provider
    CB2 --> Provider
```

Schemas

Capability Request

```
JSON
{
  "request_id": "req_109283",
  "agent_id": "code_refactor_01",
  "required_capabilities": ["coding.repository", "tool_use.parallel"],
  "preferred_capabilities": ["context.large", "cost.low"],
  "constraints": {
    "max_latency_ms": 15000,
    "max_cost_usd": 0.05,
    "privacy": "PUBLIC"
  },
  "messages": [...],
  "tools": [...],
  "reasoning_budget": "high"
}
```

Routing Decision Record

```
JSON
{
  "request_id": "req_109283",
  "eligible_deployments": ["dep_aws_claude_5", "dep_azure_gpt5"],
  "rejected_deployments": [
    {"deployment": "dep_local_vllm", "reason": "Insufficient context limit"}
  ],
  "selected_deployment": "dep_aws_claude_5",
  "expected_cost": 0.012,
```

```
  "confidence_score": 0.98,
  "fallback_chain": ["dep_azure_gpt5", "dep_gcp_gemini"]
}
```

Algorithms and pseudocode

Filtering and Scoring Algorithm (Deterministic)

Python

```
def route_request(request: CapabilityRequest, registry: Registry) -> Deployment:
    candidates = registry.get_all_active_deployments()

    # 1. Hard Constraints (Deterministic Filtering)
    eligible =
    for model in candidates:
        if not satisfies_privacy(model, request.constraints.privacy):
            continue
        if not has_required_capabilities(model, request.required_capabilities):
            continue
        if estimate_cost(model, request) > request.constraints.max_cost:
            continue
        if is_circuit_breaker_open(model):
            continue
        eligible.append(model)

    if not eligible:
        raise NoEligibleModelError("No deployment satisfies hard constraints.")

    # 2. Soft Scoring
    scores = {}
    for model in eligible:
        score = 100.0
        # Historical reliability weight
        score += registry.get_reliability_score(model.id) * 20
        # Cost deduction
        score -= (estimate_cost(model, request) / request.constraints.max_cost) * 10
        # Capability preferences
        for cap in request.preferred_capabilities:
            if has_capability(model, cap):
                score += 5
```

```
    # Cache probability adjustment
    if estimate_cache_hit_ratio(model, request) > 0.5:
        score += 15

    scores[model.id] = score

  # Sort and return highest scoring deployment deterministically
  best_deployment = max(scores, key=scores.get)
  return best_deployment
```

JSON Schema Translation (Strict Mode Emulation)

Python

```
def translate_schema_for_openai_strict(canonical_schema: dict) -> dict:
  """Mutates standard JSON schema to comply with OpenAI strict: true rules."""
  translated = deepcopy(canonical_schema)

  def walk_and_mutate(node):
    if not isinstance(node, dict): return

    # Strip forbidden defaults
    if "default" in node:
      del node["default"]

    if node.get("type") == "object":
      # Must set additionalProperties to false
      node["additionalProperties"] = False

      properties = node.get("properties", {})
      required_fields = node.get("required",)

      # Every property must be required. If it was optional, make it nullable.
      new_required = list(properties.keys())
      node["required"] = new_required

      for prop_name, prop_val in properties.items():
        if prop_name not in required_fields:
          # Convert optional type to nullable union
          current_type = prop_val.get("type")
          if isinstance(current_type, str):
            prop_val["type"] = [current_type, "null"]
```

```
            walk_and_mutate(prop_val)

    walk_and_mutate(translated)
    return translated
```

Practical scenarios

Scenario A: Strong coding task An agent requests coding.repository, tool_use.parallel, and context.large with a high reasoning budget. The gateway dynamically filters out local 8B parameter models lacking parallel tool execution. It scores cloud reasoning models, factoring in prompt caching probability (e.g., DeepSeek V4 Pro).[12] It selects the optimal model, translates the canonical tools into the provider's specific tool format, passes the reasoning_budget: "high" instruction as output_config: {"effort": "high"} for Anthropic or reasoning_effort: "high" for OpenAI [4], and successfully returns the refactored code without the agent ever knowing which physical model fulfilled the request.

Scenario B: Cheap bulk classification A monitoring fleet must classify 500,000 records. The request requires classification.basic and cost.low and permits asynchronous batching. The gateway determines that local endpoints lack the immediate concurrency required, while premium cloud reasoning models flagrantly violate the cost ceiling. It routes the payload via a provider's Batch API using a highly efficient open-weight model (e.g., DeepSeek V4 Flash) with an enforced structured output schema, achieving an effective cost of fractions of a cent per request.[12]

Scenario C: Vision task A creative-review agent needs to inspect packaging artwork. The capability request mandates vision.high_res. The gateway filters out text-only deployments. It handles the transport adaptation seamlessly, converting the canonical image payload (base64 encoded or URL) into the specific nested JSON structure required by the target provider (e.g., OpenAI's image_url object versus Anthropic's image source block), shielding the agent from multimodal API format inconsistencies.

Scenario D: Private client task An agent processing confidential financial documents sets privacy: "EU_STRICT". The gateway strictly filters deployments, removing global endpoints. It routes to an Azure OpenAI deployment explicitly configured as "Data Zone Standard" to ensure European residency.[2] If the Azure endpoint is overloaded and returns a 503 error, the fallback router attempts to escalate to a secure local model. If no local model possesses the required reasoning capability, the gateway enforces a deterministic failure rather than silently violating data residency by falling back to a US-based endpoint.

Scenario E: Long-context research A research agent uploads a 400,000-token document repository. The context-capacity manager calculates effective tokens. While several models declare 1M token limits, the router observes that DeepSeek V4 Pro offers substantial 98% discounts for cached prefix tokens.[12] It routes the request there, tracking the cache hit metrics in the telemetry ledger to optimise future routing of similar massive document sets, drastically reducing operational expenditure.

Scenario F: Provider outage Midway through a multi-step workflow, the primary cloud API

returns a 503 Service Unavailable. The provider adapter normalises this to a retryable error. The gateway attempts a single retry with exponential backoff. Upon secondary failure, the circuit breaker opens for 5 seconds.[18] The fallback router selects an equivalent capability model from a separate hyperscaler. Crucially, the capability adapter dynamically reformats the entire conversation history—including moving system prompts and translating previous tool calls—so the new provider accepts the state flawlessly, executing plugins and semantic caching fresh for the fallback node.[7]

Scenario G: Structured-output incompatibility An agent requests a highly complex, nested schema with conditional if/then logic. The gateway determines that OpenAI's strict constrained decoding rejects conditional schemas.[3] The gateway degrades gracefully: it skips native structured output, falls back to standard JSON generation via prompt injection, parses the response, and uses an internal validation loop to automatically retry and repair schema violations before returning the clean data to the agent.

Scenario H: Model replacement A next-generation model is released. The platform administrator registers it. The gateway probes its capabilities and begins shadowing 5% of traffic. Telemetry reveals its tool-calling reliability is 99%, outperforming the current production standard. The gateway automatically increments its capability scores, gradually siphoning canary traffic from older deployments until the new model safely dominates the routing pool. If anomaly detection flags safety refusals, a rollback is triggered instantaneously via control-plane policy.

Scenario I: Cloud-to-local fallback Internet connectivity to external hyperscalers fails. The gateway's circuit breakers detect universal transport timeouts. The fleet orchestrator continues running, but the gateway redirects all possible capabilities to local on-premises models. Complex reasoning tasks that cannot run locally are paused via durable queues, while basic extraction and summarisation tasks continue seamlessly on local hardware, providing degraded but highly available operation.

Scenario J: Multiple fleets A company operates a coding fleet, a localization fleet, and an HR compliance fleet sharing one central gateway. The gateway uses the tenant ID to enforce varied governance. The coding fleet receives broad access to high-cost reasoning models. The localization fleet is capped at a strict cost limit per 1000 tokens. The HR fleet is hard-locked to local, zero-data-retention deployments. All agents use identical code; only the control-plane policies differ, ensuring auditable, centralised control over decentralised fleets.

Anti-patterns

1. **Hard-coding model names into agents**: Initially appears attractive for fast prototyping. Fails immediately when models are deprecated, renamed, or experience outages. Symptoms include catastrophic workflow failure during API changes. *Safer replacement: Request logical capabilities and allow the router to resolve them.*
2. **Equating API compatibility with behavioural compatibility**: Assuming any "OpenAI-compatible" local server behaves identically. Fails due to differing tokenisers, prompt sensitivities, and fine-tuning variances. Symptoms include hallucinated arguments and schema failures. *Safer replacement: Use model-specific adapters even*

behind compatible APIs.

3. **Lowest-common-denominator interfaces**: Stripping advanced features (reasoning controls, strict schemas) to force universal compatibility. Fails by ignoring the primary advantages of advanced models. Symptoms include poor task completion despite using flagship models. *Safer replacement: Implement tiered progressive enhancement and capability adapters*.
4. **Trusting provider feature claims without testing**: Routing based on marketing documentation rather than observed capabilities. Fails when models hallucinate tool formats despite claiming support. Symptoms include high error rates on new models. *Safer replacement: Implement continuous telemetry and automated capability decay scores*.
5. **Routing only by benchmark score**: Sending every task to the most powerful model. Fails by inflating costs exponentially for trivial tasks. Symptoms include massive API bills for simple classifications. *Safer replacement: Capability requests must specify maximum cost ceilings and target quality tiers*.
6. **Raw provider errors driving workflow logic**: Letting an agent's code parse an HTTP 429. Fails when providers change error strings. Symptoms include broken retry loops. *Safer replacement: Gateway normalises errors into actionable categories (retry, fallback, fail)*.
7. **Ignoring privacy during routing**: Relying on prompt instructions (e.g., "Do not leak PII") to protect data. Fails because models cannot enforce infrastructure-level data residency. Symptoms include GDPR violations via load-balancing routing.[2] *Safer replacement: Hard infrastructure policy filters applied prior to scoring*.
8. **Executing malformed tool calls**: Allowing the provider to trigger actions based on hallucinated parameters. Fails by executing dangerous or invalid system state changes. Symptoms include database corruption. *Safer replacement: Gateway validates canonical schemas deterministically before passing control back to the agent orchestrator*.
9. **No adapter versioning**: Updating prompt logic across the board without testing. Fails by breaking older, stable fleets. Symptoms include sudden regressions in previously functioning workflows. *Safer replacement: Explicitly version model and prompt adapters, recording the version in the telemetry ledger*.

Testing strategy

A comprehensive testing framework must evaluate the gateway architecture entirely independently of live, unpredictable upstream APIs.

Adapter unit tests validate that prompt translation matrices function securely, ensuring XML formatting tags or Markdown headers are injected accurately based on model family rules.

Contract tests utilize deterministic model stubs (mock servers returning fixed, syntactically complex JSON) to prove that the unified model API contract is preserved across translation boundaries.

Structured-output tests feed deliberately malformed schema outputs from a mocked provider into the capability adapter to verify that internal retry-and-repair loops function correctly and

eventually fail deterministically if unrepairable. Token-estimation tests validate the context-budget manager by feeding massive inputs and ensuring truncation or escalation logic triggers precisely before simulated rate limits are breached.
Chaos tests randomly terminate connections or simulate HTTP 503 errors to prove the circuit breakers and capability-based fallback routers function without dropping agent intent. Finally, privacy-routing tests inject conflicting policies—such as requesting a US-only model for an EU-tagged payload—to ensure deterministic rejections occur, preventing data leakage at the unit-test level.

Evaluation scorecard

When evaluating architectural approaches (e.g., Direct Provider SDK vs Capability-Based Gateway), assess the systems using this weighted scorecard:

Criterion	Direct Provider SDK	Thin Unified Library	Central Capability Gateway
Portability	Low (Total lock-in)	Medium (API portable)	High (Model independent)
Capability Preservation	High (Native features)	Low (Lowest common denominator)	High (Progressive enhancement)
Reliability (Failover)	Low (Manual application retries)	Medium (Basic provider fallbacks)	High (Automated capability chains)
Governance / Privacy	Low (Agent dependent)	Low (Agent dependent)	High (Centralised policy enforcement)
Cost Efficiency	Low (Defaults to expensive)	Medium (Manual routing rules)	High (Dynamic capability matching)
Operability / Observability	Low (Scattered application logs)	Medium (Unified logs)	High (Unified telemetry and feedback)

Minimum viable architecture

For an advanced solo builder or small team, attempting to build a fully distributed, machine-learned gateway control plane on day one is a critical anti-pattern.
The minimum viable implementation entails establishing a robust, practical stack. First, build one unified model-call interface using standard JSON. Second, implement a configuration file mapping logical aliases (e.g., coding.high) to a small, static array of physical endpoints. Third, construct basic provider adapters covering OpenAI, Anthropic, and a local inference server. Fourth, introduce a structured error taxonomy mapping basic HTTP codes to a simple retry loop. Finally, implement a schema validation step that ensures model output strictly matches expectations before returning control to the agent.
Advanced functionality to defer includes: machine-learned routing, continuous real-time bandit routing algorithms, complex multi-objective constraint optimisation, autonomous model onboarding, and heavily distributed circuit breakers. These should be deferred until fleet scale, concurrency, and financial expenditure dictate their strict necessity.

Implementation roadmap

The architectural deployment follows a staged, low-risk roadmap.
Stage 1: Unified invocation contract Build the provider-neutral request/response schema, basic HTTP transport adapters, and fundamental error normalisation. Exit criteria: An agent can swap OpenAI for Anthropic manually via a configuration flag without any code changes.
Stage 2: Tool and structured-output normalisation Introduce the canonical tool schema. Implement the translation layer for OpenAI strict JSON mode and Anthropic input schemas. Build argument validation and schema-constrained output repair policies. Exit criteria: Agents execute reliable tool loops regardless of backend provider idiosyncrasies.
Stage 3: Logical models and capabilities Build the static capability registry and rules-based router. Replace all hard-coded model names in agent definitions with capability requests. Exit criteria: The fleet routes successfully based on alias and capability tags.
Stage 4: Reliability Introduce the error normaliser, exponential backoff, and capability-aware fallback chains. Implement deployment health tracking and circuit breakers. Exit criteria: A simulated provider outage results in seamless fallback without workflow interruption.
Stage 5: Governance Deploy the policy engine, budgeting controls, credential broker, and strict regional privacy routing. Exit criteria: EU-tagged data is deterministically blocked from US endpoints.
Stage 6: Local and cloud hybrid support Integrate local inference adapters, tracking hardware health and managing failover between edge models and cloud escalation. Exit criteria: Confidential data processes locally, escalating only when reasoning limits are breached.
Stage 7: Evaluation-driven routing Implement continuous telemetry, shadow traffic, and automated capability decay. Track task-specific performance. Exit criteria: Model routing weights adjust dynamically based on live success rates.
Stage 8: Advanced optimisation Introduce dynamic weighted routing, contextual bandits, semantic caching logic, and automated prompt optimisation. This stage is executed only when justified by scale.

Final recommended design

The optimal architecture for an advanced team managing persistent agent fleets is a rigorously segregated **Capability-Based Model Gateway**. Agents must never name physical models directly; instead, they establish a declarative contract specifying necessary capabilities (vision, reasoning, tool-use), desired constraints (latency, cost ceilings), and hard limits (privacy, region).

The universal model interface mandates canonical messages, capability lists, and budget limits, relegating all model-specific parameters to optional extension objects. Capability claims must be validated via continuous telemetry rather than static provider documentation. Tool calling and structured output schemas must be strictly normalised by the gateway, dynamically altering payloads to bypass provider idiosyncrasies (such as strict-mode defaults) while maintaining execution safety.

By normalising streaming protocols, enforcing deterministic error handling, and routing based on empirical capability profiles, the gateway shields the agent logic from the volatile infrastructure layer. This prevents the lowest-common-denominator compromise, allowing advanced fleets to exploit cutting-edge native features while gracefully falling back to robust baseline models during inevitable provider degradation.

Open questions and unresolved trade-offs

- **Cache Predictability**: While providers like DeepSeek offer substantial prompt caching discounts (reducing costs by 98%), forecasting cache hit rates prior to routing remains highly non-deterministic, complicating strict budget-based routing algorithms.
- **Reasoning Portability**: Abstracting "reasoning effort" into universal concepts remains imprecise, as models utilise inference compute radically differently. Standardising the evaluation of intermediate thinking tokens across vendors is currently an open research problem.
- **Latency vs. Emulation Overhead**: Heavy schema mutation and validation within capability adapters increases round-trip latency, posing challenges for highly real-time, voice-to-voice agent fleets that demand sub-200ms response times.

Works cited

1. What is the EU Data Boundary? - Microsoft Privacy, accessed July 25, 2026, https://learn.microsoft.com/en-us/privacy/eudb/eu-data-boundary-learn
2. How to use Azure OpenAI GDPR Compliant? - Blog by Simon Frey, accessed July 25, 2026, https://simon-frey.com/blog/how-to-use-azure-openai-gdpr-compliant/
3. OpenAI Structured Outputs in Laravel: Enforcing JSON Schema for Production AI Pipelines, accessed July 25, 2026, https://origin-main.com/laravel-architecture/laravel-openai-structured-outputs-json-schema/
4. Steering thinking - Claude Platform Docs, accessed July 25, 2026,

https://platform.claude.com/docs/en/build-with-claude/thinking-steering-and-cost
5. Thinking in tool and multi-turn workflows - Claude Platform Docs, accessed July 25, 2026, https://platform.claude.com/docs/en/build-with-claude/thinking-tool-workflows
6. Load Balancing - Router - LiteLLM, accessed July 25, 2026, https://docs.litellm.ai/docs/routing
7. Adaptive Model Routing and Fallback Logic: Routing Around LLM Provider Outages with Bifrost - DEV Community, accessed July 25, 2026, https://dev.to/kuldeep_paul/adaptive-model-routing-and-fallback-logic-routing-around-llm-provider-outages-with-bifrost-4g3m
8. Fallbacks (Provider Failover) - LiteLLM, accessed July 25, 2026, https://docs.litellm.ai/docs/proxy/reliability
9. Routing & Load Balancing - LiteLLM, accessed July 25, 2026, https://docs.litellm.ai/docs/routing-load-balancing
10. Anthropic Tool Calling - Parallel Docs, accessed July 25, 2026, https://docs.parallel.ai/integrations/anthropic-tool-calling
11. Navigating OpenAI's JSON-Structured Outputs: Limitations and Solutions - Daniel Saiz, accessed July 25, 2026, https://dsaiztc.com/blog/posts/navigating-openai-json-structured-outputs.html
12. DeepSeek pricing 2026: V4, R1, API costs, and how to optimize - CloudZero, accessed July 25, 2026, https://www.cloudzero.com/blog/deepseek-pricing/
13. Extended thinking - Claude Platform Docs, accessed July 25, 2026, https://platform.claude.com/docs/en/build-with-claude/extended-thinking
14. With OpenAI o1 & o3 Models Devs Can Set Reasoning Effort - Cobus Greyling - Medium, accessed July 25, 2026, https://cobusgreyling.medium.com/with-openai-o1-o3-models-devs-can-set-reasoning-effort-b05907c6dfa1
15. Reasoning models | OpenAI API, accessed July 25, 2026, https://developers.openai.com/api/docs/guides/reasoning
16. Extended thinking - Amazon Bedrock - AWS Documentation, accessed July 25, 2026, https://docs.aws.amazon.com/bedrock/latest/userguide/claude-messages-extended-thinking.html
17. Thinking - Claude Platform Docs, accessed July 25, 2026, https://platform.claude.com/docs/en/build-with-claude/thinking
18. LiteLLM GitHub Explained: Enterprise AI Gateway With Fallback Routing - YouTube, accessed July 25, 2026, https://www.youtube.com/watch?v=ltA-hLJofdc
19. Azure OpenAI service in europe - Microsoft Learn, accessed July 25, 2026, https://learn.microsoft.com/en-us/answers/questions/2262985/azure-openai-service-in-europe