Universal Multi-Vendor Agent Harness: Architecture and Specification

1. Core Research Objective and Harness Definition

The central objective in designing a universal, provider-neutral agent harness is to decouple the probabilistic reasoning model from the deterministic orchestration infrastructure. In the current landscape, applications frequently hardcode architectural dependencies on specific provider semantics, such as Anthropic's Messages API or OpenAI's Chat Completions API.[1] This creates an inflexible monolithic system where the foundational model serves as the architectural core rather than a replaceable execution engine.

A universal agent harness establishes an execution environment where OpenAI GPT models, Anthropic Claude models, Google Gemini models, and local open-source models operate under unified execution rules. The harness is defined precisely as the control and execution infrastructure that prepares, constrains, observes, and controls the model. It is expressly not the language model. Furthermore, the harness must be distinguished from adjacent concepts:

- **A prompt template** formats string structures; the harness orchestrates state and lifecycles.
- **An agent framework** (e.g., LangChain) dictates internal cognitive loops; the harness provides the secure infrastructure within which any cognitive loop safely executes.
- **A model gateway** (e.g., Portkey) routes HTTP requests [2]; the harness manages long-running execution, memory, and physical side effects.
- **A workflow engine** manages deterministic business logic; the harness adapts probabilistic outputs into verifiable deterministic states.
- **An MCP server** provides tool discovery and execution [3]; the harness acts as the MCP client, wrapping tool invocations in policy and security hooks.

To achieve this, the architecture separates the reasoning model from the runtime authority. The model may propose actions, but the harness retains exclusive authority to execute tools, transition state, and mutate the environment.

1.1 Architectural Principles and Invariants

The architecture is governed by specific principles, necessitating a clear delineation between non-negotiable invariants and implementation preferences.

Architectural Invariants:

1. **Deterministic Orchestration Around Probabilistic Models:** State transitions, permission evaluation, and hook executions must be strictly deterministic, neutralizing the stochastic nature of the model output.
2. **Least-Privilege Access:** Models possess zero implicit permissions. The harness must intercept and authorize every proposed external interaction.
3. **Explicit State Rather Than Hidden State:** Provider-specific conversation history must

not serve as the system of record. True state must reside in explicit memory arrays, workspace artifacts, and machine-readable execution checkpoints.
4. **Immutable Execution Evidence:** Every API call, context assembly, and tool result must be logged immutably for auditability and reproducibility.
5. **Separation of Control and Data Planes:** Routing decisions and policy validations operate independently from the physical workspace execution environments.

Implementation Preferences (Flexible):

1. *Graceful Degradation:* If an optional capability (e.g., native image understanding) is absent, the system should gracefully degrade by falling back to a text-based image-captioning tool.
2. *Controlled Provider-Specific Optimization:* The system should utilize provider-specific optimizations (like Anthropic's prompt caching or extended thinking [4]) when available, provided they are safely abstracted into an extension namespace and do not leak into the universal contract.

Table 1: Universal Runtime vs. Provider-Specific Agent Implementation

Feature Area	Provider-Specific Implementation	Universal Runtime Harness	Trade-offs
State Management	Relies on provider's conversational thread/history.	Centralized state machine and persistent database.	Universal runtime requires higher upfront engineering but eliminates vendor lock-in.
Tool Execution	Model natively triggers tools; host executes blindly.	Model proposes intent; Harness validates via security hooks before execution.	Harness adds latency per tool call but ensures deterministic security boundaries.
Memory	Opaque provider-managed memory endpoints.	Abstracted vector/semantic retrieval injected dynamically into context.	Universal requires custom deduplication and ranking, but guarantees data residency.
Error Handling	Provider-specific retry loops on	Cross-provider fallback chains and	Universal enables resilient

	429/500 errors.	idempotent replays.	long-running agents but complicates checkpoint state tracking.

2. Universal Agent Runtime Contract

A provider-neutral runtime contract must represent all significant agent interactions. The contract acts as the ultimate boundary between the application orchestration and the provider execution layer. It relies on a combination of internal typed language interfaces (e.g., TypeScript interfaces or Python Pydantic models) and JSON Schema for wire serialization and strict schema enforcement.

2.1 Canonical Data Structures

The smallest viable universal runtime contract consists of the following core entities:

AgentTask: Defines the immutable objective and constraints.

- *Required Fields:* task_id (UUIDv7), objective (string), acceptance_criteria (array of strings).
- *Optional Fields:* dependencies (array of task IDs), budget (object).
- *Lifecycle:* Owned by the control plane; immutable once transitioned to running.

AgentContext: The dynamically assembled state passed to the adapter.

- *Required Fields:* system_directives (array), messages (array of canonical message types), tools (array of MCP-compliant definitions [6]).
- *Provenance:* Context objects must carry a provenance_id mapping to their origin to prevent memory poisoning.

AgentCheckpoint: A provider-neutral snapshot enabling resumability.

- *Required Fields:* checkpoint_id, task_id, resolved_steps, pending_state, workspace_snapshot_uri.

CompletionState: The machine-readable final result.

- *Required Fields:* status (enum: success, failure), artifacts (array of URIs), validation_evidence (string).

Illustrative JSON Schema for CompletionState

JSON

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://universal-harness.internal/schemas/completion-state.json",
```

```
 "type": "object",
 "properties": {
  "task_id": { "type": "string", "format": "uuid" },
  "status": { "type": "string", "enum": ["completed", "completed_with_warnings", "failed"] },
  "deliverables": {
   "type": "array",
   "items": {
    "type": "object",
    "properties": {
     "name": { "type": "string" },
     "uri": { "type": "string" },
     "hash": { "type": "string" }
    },
    "required": ["name", "uri"]
   }
  },
  "confidence_score": { "type": "number", "minimum": 0, "maximum": 1 }
 },
 "required": ["task_id", "status", "deliverables"]
}
```

2.2 Task and Completion-State Model

To handle long-running, probabilistic workloads, the state machine utilizes a durable execution framework (e.g., Temporal.io) to ensure resumability without brittle manual state polling.[7] By decoupling the agent's logic from process memory, Temporal autosaves application state.

Valid State Transitions:

1. created → queued → preparing (workspace init) → ready
2. ready → running (model invocation)
3. running → waiting_for_tool_result | waiting_for_human_input | failed | checkpointed
4. waiting_for_tool_result → running (upon tool completion) | retrying
5. running → completed (Requires typed CompletionState)

Transitions out of running require optimistic concurrency control to prevent stale workers from duplicating events.[8] To prevent a provider from returning a persuasive natural-language response while failing to produce a required artifact, the runtime enforces that a task can only transition to completed if the model executes a predefined SubmitCompletion tool adhering to the CompletionState schema. If the model merely outputs "I am done", a before_verification hook rejects the response, appending an error message to the context instructing the model to use the proper submission tool.

Diagram 1: Task State Machine

Code snippet
```
stateDiagram-v2
  [*] --> created
  created --> queued
  queued --> preparing : Provision Workspace
  preparing --> ready
  ready --> running : Invoke Model
  running --> waiting_for_tool_result : Tool Call Proposed
  waiting_for_tool_result --> running : Tool Executed
  running --> checkpointed : Budget/Context Limit Reached
  running --> waiting_for_human_input : Approval Required
  running --> failed : Non-recoverable Error
  running --> completed : SubmitCompletion Tool Validated
  completed --> [*]
```

3. Provider Adapter SDK

The Provider Adapter SDK isolates idiosyncratic API behaviors. The architecture enforces a strict boundary:Application → Universal Runtime → Provider Adapter Interface → External Provider.

3.1 Interface Definition

Adapters are long-lived, dynamically loaded modules versioned independently from the core runtime. The minimum interface includes:

- negotiate_capabilities(AgentTask): Compares task requirements against native features.
- prepare_request(AgentContext): Translates canonical state into provider payloads.
- invoke(ProviderPayload): Returns an asynchronous stream of UniversalEvent objects.
- cancel(request_id): Maps to provider-specific cancellation semantics.
- classify_error(ProviderError): Normalizes rate limits, context overflows, and timeouts into UniversalError types.

Diagram 2: Provider Adapter Boundary

Code snippet
```
flowchart TD
  UR -->|AgentContext| PAI[Provider Adapter Interface]
  PAI --> OA[OpenAI Adapter]
  PAI --> AA[Anthropic Adapter]
```

```
PAI --> GA[Google Gemini Adapter]
PAI --> LA[Local vLLM Adapter]

OA -->|JSON Schema| OAPI[OpenAI API]
AA -->|Messages + Thinking| AAPI[Anthropic API]
GA -->|systemInstruction| GAPI[Gemini API]
LA -->|Grammar Constraints| LAPI[Local Inference]
```

3.2 Normalizing Message Hierarchies

Providers represent message hierarchies inconsistently. OpenAI utilizes distinct developer (formerly system) roles [1]; Anthropic treats system prompts as a top-level parameter distinct from the user/assistant array [1]; Google Gemini utilizes a systemInstruction parameter inside its GenerationConfig.[9]
The canonical message model preserves instruction precedence: Runtime Invariants > Organization Policies > Task Instructions > User Input.

- *Anthropic Adapter:* Concatenates high-level instructions into the top-level system string.
- *OpenAI Adapter:* Maps invariants and policies to the developer role.
- *Local Models (Missing Roles):* If a local model lacks a system role entirely, the adapter emulates it by injecting the instructions into the first user message, guarded by delimiter tokens. However, if this emulation weakens security boundaries (e.g., allowing user content to overwrite system directives via prompt injection), the adapter must reject the task and log a capability failure.

Table 2: OpenAI, Anthropic, Google, and Local Adapter Requirements

Provider	System Prompt Mapping	Tool Calling Mechanism	Structured Output Approach	Edge Cases & Quirks
OpenAI	developer role within messages array.[1]	Native tools array.[2]	response_format with strict: true.[11]	Cannot combine reasoning_effort with tool calling natively.[11]
Anthropic	Top-level system parameter.[1]	Native tools array.	Relies on tool-calling schemas for forced JSON.	thinking blocks require passing encrypted signatures

				back.[5]
Gemini	systemInstruction in request body.[10]	Native tools array inside toolConfig.[13]	response_schema / response_mime_type.[14]	Strict requirements on alternating user/model turns in history.
Local (vLLM)	Emulated in first user message if unsupported.	Grammar-constrained decoding (e.g., Outlines).	Grammar-constrained decoding.	May lack native parallel tool call support; must be serialized.

3.3 Structured Outputs and Schema Enforcement

The runtime demands machine-readable outputs for completion states and tool arguments. For OpenAI, the adapter uses strict: true JSON schema enforcement.[11] For Gemini, it uses response_schema.[14] For weaker local models, the adapter implements a parser-and-repair loop: if the output is malformed, the adapter intercepts the error, generates an internal user message containing the parser error, and queries the model again. This repair loop is hard-capped at 3 attempts to prevent infinite consumption of budget.[8]
Outputs that must *always* be machine-readable include: tool_calls, completion_state, checkpoint_metadata, and handoff_package.

4. Avoiding the Lowest-Common-Denominator Problem

A universal architecture must not reduce every provider to the smallest shared feature set. To avoid this, the architecture utilizes a two-layer approach: a **Stable Universal Core** and an **Optional Provider Extensions** namespace.
The core enforces standard message routing and tool execution. Extensions allow graceful progressive enhancement. For example, Anthropic's Claude models support extended reasoning through thinking blocks.[5] These blocks contain an encrypted signature that must be returned unmodified in multi-turn tool-use loops to maintain reasoning continuity.[5]
The adapter places this signature in AgentContext.extensions.anthropic.thinking_signature. The core runtime treats this payload opaquely, persisting it into the memory ledger without mutating it. During the next invocation, the Anthropic adapter extracts the signature and reconstructs the required API payload. This prevents provider-specific features from contaminating core orchestration logic.
Table 3: Native, Emulated, Partial, and Unsupported Capabilities

Capability	Native Implementation	Emulated Implementation	Degradation/Unsupported Action
JSON Schema	OpenAI strict: true.[11]	Parser-and-repair loop for local models.	If repair fails 3x, abort task and fallback.
Parallel Tools	Anthropic native parallel arrays.	Sequential execution by adapter intercepting parallel requests.	If interdependent, reject execution.
Thinking Blocks	Anthropic native thinking + signature.[12]	N/A (Cannot be reliably emulated).	Omit capability; use standard text generation.
Prompt Caching	Anthropic cache_control blocks.[2]	N/A	Degrade to standard pricing/latency.

5. Tool-Call Abstraction and the Model Context Protocol (MCP)

Tools are intentionally requested capabilities to inspect or change external state. The harness adopts the Model Context Protocol (MCP) [3] as the universal tool-call standard.
The universal protocol represents tool calls with attributes including tool_name, arguments, call_identifier, idempotency_key, and risk_level.
Critical Invariant: The model *never* executes a tool directly. The model returns a string indicating intent. The harness intercepts this intent, maps it to the MCP client, and triggers the lifecycle hooks.

- If a tool is *read-only* (e.g., list_directory), it executes immediately.
- If a tool is *destructive* (e.g., drop_table), the before_tool_call hook interrupts execution, changes the state to waiting_for_human_input, and serializes the context.
- If a tool returns untrusted external data (e.g., querying a public web page), the result is flagged. This triggers a middleware parser to wrap the content in XML delimiter tags, warning the model to treat the content as data, not instruction, mitigating prompt-injection risks.

6. Lifecycle Hooks and Dependency Management

Hooks are automatic lifecycle actions triggered by runtime events. They operate entirely independently of the model's awareness.

Event Catalogue (Key Hooks):

- before_workspace_prepare: Provisions microVM sandboxes.[16]
- before_context_build: Retrieves memory and filters context.
- before_model_call: Calculates cost estimates and checks token budgets.
- before_tool_call: Classifies risk, scans arguments for secrets, requests approval.
- after_tool_call: Normalizes results and detects prompt-injection payloads.

6.1 Hook Registry and Ordering

Hooks are managed via a central Hook Registry and organized by scope (global, organization, project, task). Because multiple hooks fire on a single event, execution ordering must be strictly deterministic. This is achieved via topological sorting based on an explicit dependency graph (Directed Acyclic Graph - DAG), rather than arbitrary numerical priorities.

Example Ordering for a High-Risk Tool Call (before_tool_call event):

1. resolve_tool_metadata
2. inspect_task_permissions (Depends on 1)
3. classify_command_risk (Depends on 1)
4. scan_arguments_for_secrets (Depends on 2)
5. enforce_workspace_boundaries (Depends on 3)
6. estimate_cost (Depends on 1)
7. create_git_checkpoint (Depends on 5)
8. request_human_approval (Depends on 3, 4, 5)
9. write_audit_event (Depends on all above) *(Execution is passed to the tool only after Step 9 completes successfully)*.

The runtime guarantees that a lower-priority hook (e.g., logging) cannot bypass or short-circuit a higher-priority security decision.

Diagram 3: Hook Ordering (Topological Sort)

Code snippet

```
graph TD
  A --> B(Resolve Metadata)
  B --> C(Inspect Permissions)
  B --> D(Classify Risk)
  C --> E(Scan for Secrets)
  D --> F(Enforce Workspace)
  F --> G(Create Checkpoint)
```

```
E --> H(Request Approval)
G --> H
H --> I(Write Audit Log)
I --> J
```

6.2 Hooks versus Middleware, Policies, and Tools

Mixing these abstractions creates architectural chaos. The decision framework dictates where logic belongs:

Table 4: Tools vs. Hooks vs. Middleware vs. Policies vs. Workflows

Abstraction	Definition & Agency	Example	Failure Mode / Handling
Tools	Model-requested capabilities to inspect/change external state.	execute_bash, git_commit	Normalization; model receives error to retry.
Hooks	Automatic actions triggered by runtime lifecycle events.	scan_secrets, budget_check	Fail closed; halts agent state machine.
Middleware	Request/response transformation layers.	Payload compression, JSON repair.	Transparent to the agent logic.
Policies	Declarative rules governing permissions and boundaries.	Prohibit egress outside VPC.	Hook evaluates policy; fails closed on violation.
Workflows	Explicit multi-step deterministic business processes.	Plan → Code → Test → Review.	Handled by durable execution engine (Temporal).

Ambiguous Classification Example: If the system automatically checks for leaked credentials in tool output, this is a **Hook**. If the model intentionally requests a vulnerability scan of the code it just wrote, that is a **Tool**. If the system is transforming a JSON schema to fit a provider's quirk,

that is **Middleware**.

Table 5: Synchronous vs. Asynchronous Hooks

Hook Type	Characteristics	Example Use Case	Trade-offs
Synchronous	Blocks execution pipeline until resolved.	scan_secrets, verify_permissions.	Adds latency to model turns; guarantees safety.
Asynchronous	Fires in background; does not block pipeline.	write_telemetry, update_dashboard.	Fast execution; cannot be used for security gates.

Table 6: Fail-Open vs. Fail-Closed Hook Policies

Policy	Behavior on Hook Error/Timeout	Applicable Categories
Fail-Closed	Immediately aborts the agent task; quarantines environment.	Permission enforcement, Workspace boundaries, Secret detection, Data residency checks.
Fail-Open	Logs warning and continues agent execution.	Optional telemetry, Non-critical analytics, Performance profiling.

7. Security, Workspace Isolation, and Git Hooks

AI agents executing arbitrary code pose severe infrastructure risks. Standard containerization (e.g., Docker) shares the host kernel, leaving the platform vulnerable to container escapes and privilege escalation.[17]

7.1 Workspace Sandboxing

The harness mandates hardware-level isolation using Firecracker microVMs (via platforms like E2B or Modal).[16] MicroVMs provide sub-second cold starts [17] while running a dedicated kernel per tenant.

Permissions are managed via capability tokens. During before_workspace_prepare, the harness provisions a microVM. Network egress is blocked by default, permitting only allow-listed API

endpoints.

7.2 Git Worktrees and Checkpoints

For coding agents (e.g., SWE-bench evaluations [19]), isolating modifications is critical. The harness provisions one Git worktree per agent task, allowing parallel execution without branch collisions.[21]

Hooks such as before_file_write enforce path ownership, ensuring the agent cannot write outside its designated /workspace boundary. before_git_commit hooks automatically run pre-commit tests.

A GitCheckpoint object is recorded iteratively. It contains: repository_id, worktree, base_commit, and a patch payload of dirty_state. This allows the exact file-system state to be snapshotted and restored seamlessly.[18]

Diagram 4: Security Trust Boundaries

```
Code snippet
flowchart TD
  subgraph Control Plane
    PR
    HR
    CR
  end

  subgraph Execution Plane
    UR
    PA[Provider Adapters]
  end

  subgraph Untrusted Sandbox
    Workspace
    ToolExec
  end

  Control Plane -- Policies & Config --> Execution Plane
  Execution Plane -- Secure RPC/MCP --> Untrusted Sandbox
  Untrusted Sandbox -- Network Egress Blocked --> Internet
```

Table 7: In-Process Hooks vs. Isolated Hook Workers

Implementation	Latency	Security Isolation	Complexity	Best For
In-Process	Low (<5ms)	None (Shares memory with runtime).	Low	Fast permission checks, context truncation.
Isolated Workers	High (RPC overhead)	High (Separate process/container).	High	Parsing untrusted tool outputs, executing external scripts.

8. Memory Integration and Context Assembly

The universal harness abstracts memory to prevent provider-specific conversation endpoints (like OpenAI's native threads) from becoming the only persistent memory mechanism.

8.1 Memory Hooks

Memory is segmented into conversation memory, episodic memory, and semantic knowledge. Hooks (before_memory_read, after_memory_write) manage interaction with a vector database. During before_memory_write, data is checked for encryption and tenant isolation requirements. A critical risk is *memory poisoning*, where an agent ingests malicious data that later compromises future context assemblies. To mitigate this, retrieved memory is strictly typed with a confidence and provenance score.

8.2 Dynamic Context Assembly

Context is assembled dynamically via a pipeline during before_context_build. The pipeline prioritizes tokens:

1. Runtime Invariants (Highest precedence)
2. Task Objective
3. Retrieved Semantic Memory (Filtered by relevance scoring)
4. Execution History / Current State
5. Tool Descriptions

If the assembled context exceeds the model's context_budget_limit, a before_context_compression hook truncates historical execution logs or utilizes a cheaper model to summarize previous episodic memories. The system immutably records what context

was selected and why, appending token costs to the execution trace.

Diagram 5: Context Assembly Flow

Code snippet

```
graph LR
  A --> E(Context Builder Pipeline)
  B --> E
  C[Organization Policies] --> E
  D --> E
  E --> F{Check Context Limit}
  F -->|> Limit| G
  G --> H[Final AgentContext Array]
  F -->|< Limit| H
  H --> I
```

9. Capability-Based Routing and Economics

Hardcoding models (e.g., model="gpt-4o") makes agent orchestration brittle. The harness utilizes Capability-Based Routing, where a task requests capabilities (e.g., minimum_context: 200000, structured_output: strict, latency: <2000ms).

9.1 Model Capability Registry

The Registry tracks models across multiple dimensions. Data is sourced from benchmark suites, provider API documentation, and historical production traces. Key fields include: model_family, context_limit, tool_call_support, latency_profile, and cost_profile.
Crucially, the registry distinguishes between a model architecture (Llama 3) and an API endpoint or hardware deployment (Llama 3 FP16 on H100 vs. Llama 3 4-bit quantized on Edge). These distinct endpoints possess vastly different capability scores.

9.2 Routing Algorithms

Static rule-based routing is insufficient for complex fleets. The harness implements Contextual Dueling Bandits to optimize routing dynamically.[22] The algorithm projects query embeddings to assess the query's complexity, balancing the expected reward (success rate) against the cost and latency overhead.[23]
Phase-based routing allows a task to switch models dynamically. A cheap model classifies the intent; a strong model performs logical decomposition; a highly specialized coding model implements the fix; and a local model processes sensitive PII data.[23]

Table 8: Static Routing vs. Dynamic Routing

Feature	Static Routing	Dynamic (Contextual Bandit) Routing
Mechanism	Hardcoded if/else rules based on task type.	Machine learning models predict success vs. cost dynamically.[22]
Overhead	Minimal compute overhead.	Requires embedding projection and bandit calculation latency.[24]
Adaptability	Fails if provider API degrades or prices change.	Automatically shifts traffic away from degraded or expensive models.[22]

9.3 Cost-Control Hooks

Cost governance operates universally. A before_model_call hook tracks real-time_usage_estimation. If a task exceeds its task_budget, an on_budget_warning event fires. The router evaluates the remaining budget and may choose to:

- Downgrade the model to a cheaper variant.
- Disable optional verification hooks.
- Pause the task and request human approval for additional budget.

10. Checkpoints, Resumability, and Handoffs

Durable execution requires robust serialization. The harness defines a universal AgentCheckpoint format independent of provider state.

10.1 Checkpoint Architecture

A checkpoint captures enough deterministic state for *another* model or provider to resume the task. It includes the task_identity, resolved steps, remaining plan, memory references, and a snapshot of the Git worktree. It explicitly *excludes* provider-private hidden reasoning that cannot be verified.

Event-sourced checkpoints (recording a ledger of actions) are preferred over snapshot checkpoints (saving the whole RAM state) because event ledgers are portable across different foundational models.

Table 9: Snapshot Checkpoints vs. Event-Sourced Checkpoints

Characteristic	Snapshot Checkpoints	Event-Sourced Checkpoints

		(Recommended)
Mechanism	Dumps the exact memory/context state of the agent framework.	Replays the ledger of validated actions and tool results.
Portability	Highly provider-specific; cannot easily map Claude state to GPT-4.	Provider-neutral; can be translated into any context format.
Debugging	Difficult to inspect intermediate reasoning logic.	Highly auditable; every state transition is an explicit event.

10.2 Universal Agent Handoffs

When a task requires specialized review (e.g., a Gemini agent reviewing a Claude agent's code), an AgentHandoff contract is generated. The orchestrator mediates this handoff asynchronously. The handoff package contains: source_agent, destination_capability, completed_work, validation_status, and the GitCheckpoint URI. Because the handoff relies exclusively on the universal contract, the destination agent can construct its own optimal context window without inheriting the source agent's conversational quirks.

Diagram 6: Cross-Provider Handoff

Code snippet

```
sequenceDiagram
    participant Orchestrator
    participant Claude (Source)
    participant ContextBuilder
    participant Gemini (Dest)

    Orchestrator->>Claude: Assign Task
    Claude->>Claude: Execute Tools & Edit Code
    Claude->>Orchestrator: Emit Handoff (Review Needed)
    Orchestrator->>ContextBuilder: Generate Universal Checkpoint
    ContextBuilder->>Gemini: Build Context from Checkpoint
    Gemini->>Gemini: Verify Code & Run Tests
    Gemini->>Orchestrator: Emit CompletionState
```

11. Fallback, Migration, and Observability

11.1 Retries and Provider Fallback

Transient provider errors (e.g., 429 Rate Limits) trigger exponential backoff. Permanent errors (e.g., invalid structured output after 3 repair loops, context overflow, or policy rejection) trigger a fallback chain.
Fallback Chain: Preferred Model (Claude 3.5) → Same-Provider Equivalent (Claude 3 Opus) → Cross-Provider Equivalent (GPT-4o) → Reduced Capability (GPT-4o-mini) → Local Model (Llama 3).
To preserve idempotency during fallback, Temporal replays previous read-only tool results directly from the event history. Destructive actions already executed are not repeated; only the remaining pending state is passed to the fallback provider.[7]

Table 10: Same-Provider Retry vs. Cross-Provider Fallback

Aspect	Same-Provider Retry	Cross-Provider Fallback
Primary Use Case	Transient API timeouts, 429 rate limits, minor JSON repair.	Hard outages, context limits exceeded, persistent safety filter rejections.
State Preservation	Can often reuse session IDs or cached prompt blocks.[2]	Requires full context reconstruction from Universal Checkpoint.
Complexity	Low (Handled at Adapter level).	High (Handled by Orchestrator and Routing Engine).

Diagram 7: Provider Fallback Sequence

Code snippet
graph TD
A --> B{API Outage or Hard Failure}
B -->|Yes| C
C --> D[Load Last Valid Checkpoint]

D --> E
E --> F
F --> G

11.2 Migration Strategy

Replacing a model requires zero changes to the underlying tools, hooks, or task logic. The migration runbook mandates:

1. Update Provider Adapter SDK logic.
2. Register the new model in the Capability Registry.
3. Run *Simulated Conformance Tests* (offline fixtures validating canonical messages and schemas).
4. Run *Shadow Traffic* (mirroring production requests to the new model without using its output).
5. Enable fallback-only use, then update Contextual Bandit weighting to permit live routing.[22]

11.3 Observability and Tracing

Logging and tracing hooks correlate actions via identifiers: trace_id, task_id, model-call_id. Standard metrics gathered include schema-valid_response_rate, retries_per_task, and checkpoint_recovery_rate. To protect privacy, a before_logging hook scans and redacts PII and proprietary reasoning traces before emitting data to external observability platforms.

12. Local Model Integration

Local models present unique challenges. Deployments via vLLM or Ollama often possess different constraints than cloud APIs.
The adapter for a local model must frequently emulate missing features (like native tool calling) using grammar-constrained generation.[14] Local cost accounting is calculated by measuring GPU infrastructure uptime amortized across tokens processed.

Table 11: Cloud vs. Local Model Execution

Feature	Cloud API (OpenAI/Anthropic)	Local Model (vLLM/Llama)
Latency	Network bound; subject to public queues.	Compute bound; predictable if hardware is dedicated.
Privacy / Security	Data leaves the perimeter; relies on BAA/DPA.	Absolute data residency; suitable for extreme

		high-risk PII.
Adapter Complexity	Low (APIs are mature and feature-rich).	High (Must emulate tools via strict grammar constraints).

13. Provider Conformance Testing

To detect adapters that appear functional but silently weaken semantics (e.g., dropping image inputs without error, or serializing parallel tool calls), a mandatory Conformance Test Suite acts as an absolute gatekeeper for any new model introduction.

Canonical Test Fixtures:

- *Strict Schema Compliance:* Forcing the model to output deeply nested JSON to test the adapter's repair loop.
- *Security Boundary Test:* Injecting prompt-injection payloads into simulated tool outputs to ensure the adapter correctly flags and isolates untrusted data.
- *Context Truncation:* Pushing context to 105% of the declared limit to verify the before_context_compression hook executes gracefully rather than resulting in a hard API crash.
- *Cross-Provider Resume:* Bootstrapping a task on OpenAI, pausing it, and verifying Anthropic can resume execution seamlessly from the universal checkpoint.

14. Architecture: Control Plane vs. Execution Plane

The system must be deployed as a layered architecture to ensure security and scalability.

- **Control Plane (High Trust):** Responsible for the Fleet Registry, Model Routing, Budgets, Policy definitions, and Audit records. It maintains the centralized Temporal cluster for workflow orchestration.[7]
- **Execution Plane (Medium/Low Trust):** Responsible for Context Assembly, Provider Invocation, and local Worker instances. It dynamically provisions isolated E2B Firecracker microVMs for actual tool and code execution.[16]

Credentials for external APIs never leave the Control Plane. The Execution Plane receives short-lived, task-scoped JWTs to interact with the orchestration layer.

Diagram 8: Control Plane and Execution Plane

Code snippet

```
graph TD
  subgraph Control Plane
```

```
    Router
    Pol[Policy Engine]
    Temp
end

subgraph Execution Plane Fleet
    W1
    W2
end

subgraph Sandbox Infrastructure
    E2B[Firecracker microVMs]
end

Temp -->|Task Lease| W1
Router -->|Model Selection| W1
Pol -->|Permissions| W1
W1 -->|Mounts Worktree| E2B
```

Table 12: Modular Monolith vs. Microservices vs. Event-Driven Architecture

Architecture	Complexity	Reliability	Scalability	Recommendation
Modular Monolith	Low	Low (Process crash kills active tasks).	Vertical only.	Good for initial prototyping.
Microservices	High	Medium (Network partitions cause state loss).	Horizontal.	Anti-pattern for agent state.
Event-Driven (Temporal)	Medium	Very High (Durable state, automatic resumes).[8]	High	**Target Architecture** for robust long-running agents.

15. Build-versus-Buy Analysis

Delegating non-core components to established technologies is critical for operational stability.

- **Workflow Engine:** *Buy* (Temporal.io). Building a durable execution state machine from scratch is a massive architectural risk.[8]
- **Secure Sandboxing:** *Buy* (E2B or Modal). Managing Firecracker microVMs, network egress rules, and sub-second boot times requires specialized infrastructure expertise.[16]
- **Tool Protocol:** *Buy/Adopt* (Model Context Protocol). Adopting an open standard prevents locking into proprietary tool server architectures.[3]
- **Universal Runtime & Hooks Layer:** *Build*. This is the strategic intellectual property of the organization. No off-the-shelf product currently provides multi-vendor hook isolation to this exact specification.

16. Threat Model and Failure-Mode Analysis

Threat Model Summary:

- *Workspace Escape:* An agent attempts to execute malicious bash commands. *Control:* E2B Firecracker isolation prevents host kernel access. before_tool_call hooks enforce directory path boundaries.
- *Prompt Injection via Tool Output:* A web search tool returns malicious instructions. *Control:* The adapter middleware sanitizes output, wrapping it in delimiter tags (e.g., <untrusted_data>) and lowering its priority in the context assembly pipeline.
- *Hook-Order Manipulation:* An attacker attempts to bypass a security scan. *Control:* Topological sorting of the Hook DAG prevents lower-priority hooks from executing before higher-priority dependencies resolve.

Failure-Mode Analysis:

- *Router Misclassification:* The capability registry assigns a complex task to a weak model. *Detection:* Task state hits failed repeatedly or completed_with_warnings. *Fallback:* Contextual bandit automatically degrades the model's score; workflow retries with the next highest-scoring model.
- *Checkpoint Corruption:* A worker crashes mid-write. *Detection:* Temporal activity timeout.[25] *Recovery:* Rollback to the previous immutable event in the event-sourced ledger.

17. Final Synthesis and Implementation Blueprint

The Universal Multi-Vendor Agent Harness internalizes state, memory, security, and lifecycle management within the orchestrating infrastructure, governed by durable execution (Temporal) and hardware-level sandboxing (E2B Firecracker).

17.1 Implementation Roadmap

1. **Phase 1: Minimum Viable Prototype (MVP):** Define the AgentTask and CompletionState

JSON schemas. Build stateless OpenAI and Anthropic adapters. Implement a hardcoded router and a basic before_tool_call security hook. Run a simple code-modification task locally.

2. **Phase 2: Production Hardening:** Integrate Temporal.io for durable state transitions. Deploy E2B for secure worktree sandboxing. Implement the capability registry and capability-based routing algorithms.
3. **Phase 3: Multi-Project Fleet:** Introduce cross-provider checkpoint resumption. Deploy phase-based routing (e.g., GPT-4o mini for planning, Claude 3.5 for execution). Implement rigorous conformance testing for dynamic adapter upgrades.

17.2 Pseudocode: Main Execution Loop (Temporal Workflow)

```
Python
@workflow.defn
class UniversalAgentWorkflow:
  @workflow.run
  async def run(self, task_input: AgentTask) -> CompletionState:
    # 1. Resolve Capabilities & Route
    model_selection = await workflow.execute_activity(
      route_task, task_input, schedule_to_close_timeout=timedelta(seconds=5)
    )

    # 2. Provision Sandbox
    workspace = await workflow.execute_activity(
      prepare_workspace, task_input.task_id,
schedule_to_close_timeout=timedelta(seconds=30)
    )

    while True:
      # 3. Assemble Context
      context = await workflow.execute_activity(build_context, workspace, task_input)

      # 4. Invoke Model (via Adapter)
      response = await workflow.execute_activity(
        invoke_provider, context, model_selection,
        retry_policy=RetryPolicy(maximum_attempts=3)
      )

      # 5. Process Intent
      if response.intent == "submit_completion":
        evidence = await workflow.execute_activity(run_verification_hooks, response)
```

```
            if evidence.is_valid:
                return CompletionState(status="completed", deliverables=response.artifacts)
            else:
                await workflow.execute_activity(log_validation_failure, evidence)
                continue # Loop back to model for repair

        elif response.intent == "tool_call":
            # 6. Execute Security Hooks and Tool
            await workflow.execute_activity(execute_hook_dag, "before_tool_call", response)
            tool_result = await workflow.execute_activity(execute_tool_in_sandbox, response,
workspace)
            await workflow.execute_activity(record_checkpoint, tool_result)
```

By strictly adhering to the Universal Runtime Contract, normalizing messages via robust Provider Adapters, standardizing tools through the Model Context Protocol, and intelligently allocating workloads via contextual dueling bandits, the harness achieves true provider neutrality. It successfully designs an operating layer where tasks transition seamlessly between GPT, Claude, Gemini, local models, or future providers without altering task definitions, tools, memory, or security policies.

Works cited

1. Anthropic vs OpenAI: Developer Experience Compared, accessed July 25, 2026, https://www.developersdigest.tech/blog/anthropic-vs-openai-developer-experience
2. OpenAI Responses API vs. Chat Completions vs. Anthropic Messages API - Portkey, accessed July 25, 2026, https://portkey.ai/blog/open-ai-responses-api-vs-chat-completions-vs-anthropic-anthropic-messages-api/
3. Specification - Model Context Protocol, accessed July 25, 2026, https://modelcontextprotocol.io/specification/2025-11-25
4. Anthropic Messages API - Abacus.AI, accessed July 25, 2026, https://abacus.ai/help/developer-platform/route-llm/anthropic-messages
5. Thinking - Claude Platform Docs, accessed July 25, 2026, https://platform.claude.com/docs/en/build-with-claude/thinking
6. Tools - Model Context Protocol, accessed July 25, 2026, https://modelcontextprotocol.io/specification/draft/server/tools
7. Durable Execution Platform | Temporal, accessed July 25, 2026, https://temporal.io/product
8. Temporal: Durable Execution Solutions, accessed July 25, 2026, https://temporal.io/
9. Function calling with the Gemini API - Interactions API | Google AI for Developers, accessed July 25, 2026, https://ai.google.dev/gemini-api/docs/function-calling
10. Gemini Interactions API - Google AI for Developers, accessed July 25, 2026,

https://ai.google.dev/api/interactions-api
11. Migrate to the Responses API - OpenAI Developers, accessed July 25, 2026, https://developers.openai.com/api/docs/guides/migrate-to-responses
12. anthropic-messages-api-openapi.yml - GitHub, accessed July 25, 2026, https://github.com/api-evangelist/anthropic/blob/main/openapi/anthropic-messages-api-openapi.yml
13. Generate content with the Gemini API - Google Cloud Documentation, accessed July 25, 2026, https://docs.cloud.google.com/gemini-enterprise-agent-platform/reference/models/inference
14. Taming the Wild Output: Effective Control of Gemini API Response Formats with response_schema | by Kanshi Tanaike | Google Cloud - Medium, accessed July 25, 2026, https://medium.com/google-cloud/taming-the-wild-output-effective-control-of-gemini-api-response-formats-with-response-schema-ae0097b97502
15. The hero's journey to AI durability with Temporal, accessed July 25, 2026, https://temporal.io/blog/the-heros-journey-to-ai-durability-with-temporal
16. E2B Review: AI Agent Sandbox Pricing, Code Interpreter, MCP, Security, and Alternatives, accessed July 25, 2026, https://aiidelist.com/ide/e2b
17. E2B Alternatives: Best Sandbox Environments for 2026 | Blaxel Blog, accessed July 25, 2026, https://blaxel.ai/blog/e2b-alternatives-sandbox-environments
18. E2B vs Modal: comparing AI code execution sandboxes in 2026 | Blog - Northflank, accessed July 25, 2026, https://northflank.com/blog/e2b-vs-modal
19. Best Sandboxes for SWE-Bench-Style Coding Agents in 2026 | Modal Blog, accessed July 25, 2026, https://modal.com/resources/best-sandboxes-swe-bench-coding-agents
20. Getting Started - SWE-agent documentation, accessed July 25, 2026, https://swe-agent.com/latest/
21. Git Worktrees for Parallel AI Agents: 2026 Guide, accessed July 25, 2026, https://noqta.tn/en/blog/git-worktrees-parallel-ai-coding-agents-guide-2026
22. LLM Routing with Dueling Feedback - OpenReview, accessed July 25, 2026, https://openreview.net/forum?id=zf3gNrp3jp
23. Learning to Route LLMs from Implicit Cost-Performance Preferences via Meta-Learning, accessed July 25, 2026, https://arxiv.org/html/2606.06178v1
24. Adaptive LLM Routing under Budget Constraints - arXiv, accessed July 25, 2026, https://arxiv.org/html/2508.21141v1
25. What are Agentic AI Workflows? Scalable & Durable Workflows - Temporal, accessed July 25, 2026, https://temporal.io/blog/build-resilient-agentic-ai-with-temporal