Canonical Protocol and Schema Catalogue for a Provider-Independent Agentic Platform

1. Primary Research Objective and Core Architectural Principle

The primary objective of this architectural research programme is to definitively establish the stable, versioned, provider-neutral contracts required to ensure that agents, models, providers, tools, workflows, execution environments, and observability systems can interoperate without sharing hidden state or vendor-specific semantics. The resulting architecture guarantees that the orchestration platform remains entirely independent of proprietary model capabilities or ephemeral transport sessions.

The foundational control principle governing this architecture dictates that natural language must be treated strictly as content, never as the production coordination protocol. Natural language may effectively describe a system's intent, appearing inside objectives, rationales, summaries, human-facing explanations, evidence excerpts, and unresolved questions. However, it is explicitly prohibited from serving as the sole representation of task identity, state, ownership boundaries, deadlines, budgets, permissions, approval requirements, acceptance criteria, completion status, tool authority, side effects, checkpoint state, artefact identity, error classification, release evidence, or protocol version.

Relying on natural-language chat as a production coordination protocol introduces unmanageable brittleness and compromises determinism. Structured contracts must govern execution. Durable state must remain outside of the model's conversational context. An orchestration platform that embeds critical state transitions within provider-specific message histories effectively surrenders control of workflow continuity to an external provider's transient context window, rendering checkpointing and cross-provider resumption impossible.

2. Foundational Hypotheses and Architecture Validation

The architecture validates several critical hypotheses that dictate the design of the Agent Platform. No single external protocol solves every interoperability layer; therefore, a composite architecture is required. Canonical internal schemas are rigorously separated from transport encoding. Provider-native request and response formats must terminate at adapter boundaries, ensuring that provider-specific fields do not contaminate portable task or workflow definitions.

Durable workflow state must not live only in prompts or provider sessions. Commands and

events demand completely different delivery and retry semantics, necessitating different protocol primitives. Tool intents and tool receipts are treated as separate records to isolate the proposal of an action from its execution. Large artefacts are exchanged strictly by cryptographic reference rather than embedding massive binaries into task payloads. Furthermore, task completion is typed and criterion-based, eliminating ambiguous binary completion states.

To maintain ecosystem stability, schema validation occurs at the perimeter before costly model execution. Every protocol record carries an explicit schema and protocol version. Old and new protocol participants coexist through strict compatibility policies and translation adapters. Additive schema changes are treated cautiously, and unknown fields are subjected to explicit stripping rules to prevent payload poisoning. Open enums are utilized for diagnostic categories, while closed enums govern strict state transitions requiring explicit version negotiation. Provider extensions utilize tightly controlled namespaces, preventing arbitrary vendor-specific fields from polluting the core. Finally, contract tests are mandated across every producer-consumer boundary, ensuring that security-critical schema changes can trigger immediate, forced migration without catastrophic system failure.

3. Interoperability Problem Map

An exhaustive architectural design must isolate and resolve interoperability domains independently. Attempting to force a single protocol to govern transport, state, and identity invariably yields brittle abstractions.

3.1 Model and Provider Transport Interoperability

Model interoperability is achieved by normalising the semantic intent of prompts, roles, structured outputs, and capability negotiation. The platform requests abstract capabilities (e.g., structured_output, vision_input, long_context, strong_code_reasoning, low_latency_classification) rather than referencing physical model names in portable workflow definitions. Adapters intercept these capability requests and project them into model-specific optimizations.

Provider transport interoperability addresses differences in HTTP endpoints, WebSockets, Server-Sent Events, authentication, and retry hints. The platform requires a normalized provider transport lifecycle model where transport errors (e.g., HTTP 429 Rate Limits) trigger deterministic backoff algorithms within the adapter boundary rather than bleeding into the durable workflow state as immediate task failures. Provider-specific features are supported via controlled extension namespaces without contaminating the portable core.

3.2 Tool, Task, and Handoff Coordination

Tool interoperability requires a rigid separation between proposing an action and executing an action. An agent generates a ToolIntent. This intent is intercepted by a deterministic policy gateway which evaluates authorization requirements, scopes, and budget limits. Only after successful validation does the adapter trigger execution, subsequently yielding a ToolReceipt. This division ensures that malicious or hallucinated tool parameters generated by a model are

quarantined before external side effects materialize.
Coordinating tasks requires a canonical protocol distinguishing assignment, delegation, escalation, and completion. Unlike JSON-RPC or REST which optimize for synchronous client-server communication, the Agent Platform requires durable, long-running, asynchronous handoff contracts. A handoff packet completely encapsulates the task identity, objective, budget, deadline, context references, and cryptographic provenance, allowing a receiving agent or provider to resume work without querying the original provider's proprietary message session.

3.3 Event, Workflow, and Checkpoint Portability

Events record immutable historical facts and must not be confused with instructions to make something happen. Workflow portability is supported by modelling canonical semantics for task graphs, dependencies, timers, retries, and compensation. However, engine-specific features (such as deterministic replay constraints or proprietary search attributes) cannot be made fully portable without excessive abstraction, and are thus wrapped behind localized interfaces.
A resilient agentic workflow survives worker crashes, platform upgrades, and provider outages. Checkpoints are operational state markers storing completed work, remaining assumptions, task identity, and context manifests. Checkpoints explicitly exclude unrestricted chain-of-thought tokens or ephemeral process memory, as these elements tie the workflow irretrievably to a specific environment.

3.4 Memory, Identity, and Telemetry Portability

Canonical memory is distinguished from derived search indexes. Canonical memory includes content, provenance, access classification, valid time, and contradiction links. Embeddings and provider-specific caches are treated as derived data; relying on provider-specific vector embeddings as authoritative memory is strictly prohibited, as switching embedding models renders previous vector spaces incompatible.
Identity portability is governed by OAuth 2.x and OpenID Connect (OIDC), ensuring that an agent role, instance, provider, and credential holder are never treated as the same identity. Telemetry portability relies on W3C Trace Context and OpenTelemetry, ensuring that traces are correlated across model, tool, workflow, Git, and release operations without losing causation lineage.[1]

3.5 Artefact, Git, and Software-Delivery Interoperability

Artefact provenance relies on content-addressable storage, cryptographic digests, OCI artefacts, and in-toto statements.[2] Transferring large outputs occurs entirely via durable artefact references, preserving lineage through content hashes rather than placing complete files into conversational context. Git and software-delivery interoperability map code changes, tests, and reviews directly to exact, immutable commit hashes, specifically rejecting mutable branch names as valid release evidence.

4. Standards and Official Specifications Programme

The orchestration platform integrates established external standards to prevent the reinvention of distributed-systems protocols. These standards are strictly bounded to the problem spaces they natively solve.

4.1 JSON Schema and OpenAPI

The canonical JSON Schema dialect for the target system is JSON Schema 2020-12. This dialect supports strict structural validation prior to expensive model inference. Within the target system, all entities require strict $schema and $id declarations. To balance forward compatibility with rigorous security, core platform envelopes utilise closed objects (additionalProperties: false) to prevent schema confusion and malicious payload injection, while explicitly defined extension namespaces permit additive flexibility. OpenAPI 3.1 is utilized exclusively for synchronous REST-based gateways and client generation, but is explicitly recognized as insufficient for asynchronous task state, durable handoffs, or event replay.

4.2 AsyncAPI and CloudEvents

AsyncAPI 3.0.0 governs queue messages, broker bindings, and asynchronous API documentation.[3] CloudEvents version 1.0.2 serves as the mandatory transport envelope for asynchronous factual notifications.[5] CloudEvents decouple the event schema from the underlying transport protocol (e.g., HTTP, AMQP, Kafka).[5] Canonical attributes include id, source, specversion, type, and time.[7] CloudEvents are strictly reserved for representing immutable historical facts and are expressly forbidden from being used as imperative commands, which require distinct synchronous validation and authorization mechanisms.

4.3 OpenTelemetry and W3C Trace Context

Distributed tracing utilizes the W3C Trace Context specification, recognized as the definitive standard for propagating trace identity across service boundaries.[1] The architecture mandates the propagation of traceparent (comprising format version, trace ID, parent ID, and trace flags) and tracestate across every component boundary.[1] The Level 2 Candidate Recommendation is adopted to ensure trace ID uniqueness, incorporating the random trace ID flag to strengthen cryptographic guarantees.[9] Trace context must be deliberately regenerated at specific trust boundaries, such as crossing disparate organizational tenants, to prevent context poisoning. The OpenTelemetry (OTel) GenAI Semantic Conventions standardize observability for LLM applications.[10] As of 2026, the GenAI conventions remain in Development status without a finalized stabilization timeline.[10] The platform adopts stable root spans (gen_ai.operation.name set to chat or embeddings) and explicitly separates provider identifiers (gen_ai.provider.name) from requested models (gen_ai.request.model) and actual responding models (gen_ai.response.model).[10] Because OTel payload capture for prompts presents security risks, it is explicitly opt-in and handled distinctly from canonical task state.[12] OTel execution telemetry is rigorously distinguished from semantic business evaluation and architectural conformance

tracking.

4.4 Model Context Protocol (MCP) and Agent-to-Agent Protocols

The Model Context Protocol (MCP) is evaluated strictly as an agent-to-tool boundary specification.[14] MCP provides robust mechanisms for tool discovery, prompt formatting, and local resource integration. However, the MCP specification dictates that descriptions of tool behavior must be considered untrusted unless obtained from a trusted server, and intentionally limits server visibility into prompts.[14] Crucially, MCP does not solve durable task orchestration, workflow checkpointing, distributed retries, or complex handoff semantics. Consequently, MCP interactions are securely wrapped within the platform's ToolIntent and ToolReceipt constructs. Agent-to-agent (A2A) protocols are exposed only at external gateways, mapped securely into the internal canonical task model to prevent untrusted external agents from dictating internal state transitions.

4.5 EU AI Act Logging and Compliance Standards

To support high-risk agentic workloads, the target system integrates automated logging controls aligned with the EU AI Act (Regulation 2024/1689). Article 12 requires high-risk systems to automatically record events over their entire lifetime, serving regulatory purposes including risk identification (Article 79), post-market monitoring (Article 72), and deployer oversight.[15] Article 26(6) mandates a minimum retention period of six months for these logs, establishing a statutory baseline that supersedes generic GDPR "necessity" arguments for operational AI logs.[15] The platform maps these requirements to ISO 42001 governance controls, ensuring every input, output, system update, and exception is logged immutably, centrally vaulted, and cryptographically timestamped.[17] Manual operator recording is strictly invalid under this framework.

5. Canonical Communication Primitives and Routing

To eliminate ambiguity, the platform classifies all inter-component interactions into a definitive catalogue of communication primitives.

Primitive	Meaning	Authoritative State	Idempotency Requirement
Message	Human-readable information with no formal state effect.	No	No
Task	Stateful unit of work with a measurable objective and acceptance	Yes	Yes (at lifecycle boundaries)

	conditions.		
Command	Authorised instruction to perform a specific action or mutation.	Yes	Yes (Mandatory)
Request	Query for information without intended state mutation.	No	N/A (Safe operation)
Response	Structured answer to a request.	No	N/A
Event	Immutable historical fact that occurred (routed via CloudEvents).	Yes	N/A (Producer responsibility)
Notification	Routed informational event alerting a subscriber.	No	No
Signal	Low-level runtime indicator (e.g., heartbeat, lease renewal).	No	Yes
Observation	Recorded perception or passive tool output.	Yes	No
Status Update	Intermittent report of task or workflow progress.	Yes	Yes
Result	Structured terminal or intermediate task	Yes	Yes

	output.		
Artefact	Durable, versioned file or object referenced by content digest.	Yes	Implied by cryptographic hash
State Transition	Authoritative change to the workflow state machine.	Yes	Yes
Handoff	Transfer of task ownership and continuation state.	Yes	Yes
Delegation	Assignment of a subtask while the parent retains accountability.	Yes	Yes
Escalation	Transfer of ownership to a higher authority due to failure or policy breach.	Yes	Yes
Approval Req.	Request for an authorised human or service policy decision.	Yes	Yes
Approval Dec.	Signed approval, rejection, or conditional approval.	Yes	Yes
Checkpoint	Durable continuation state enabling	Yes	Yes

	cross-provider resumption.		
Incident Record	Structured record of operational, safety, or security failure.	Yes	Yes

Formal Routing Decision Tree:

1. If the interaction involves long-running state, ownership, and strict acceptance criteria, it must be routed as a **Task**.
2. If the interaction instructs an immediate, specific side-effect, it must be routed as a **Command**.
3. If the interaction broadcasts a completed historical fact to multiple consumers, it must be routed as an **Event** mapped to the CloudEvents 1.0.2 schema.[5]
4. If the data exceeds optimal token/payload limits or requires strict immutability guarantees, it must be exchanged via an **Artefact Reference**, retaining the actual payload in content-addressable storage.
5. If the communication is primarily human-readable and no machine-authoritative transition depends on parsing it, it is a **Message**.

6. Canonical Schema Catalogue

The platform's logical architecture relies on standardizing JSON Schemas across 13 distinct domains, establishing the structural foundation for provider independence.

6.1 Core Domains Overview

- **Organisation and project domain:** Governs multi-tenant isolation, defining Organisation, Workspace, Project, ProjectPolicy, and ProjectBudget.
- **Agent and fleet domain:** Defines logical capabilities, including AgentRole, AgentCapability, AgentInstance, and AgentAssignment.
- **Task and workflow domain:** The operational core, encompassing WorkflowDefinition, AgentTask, TaskAttempt, CompletionState, and AcceptanceCriterion.
- **Execution and continuity domain:** Manages resilience via AgentCheckpoint, AgentExecutionRecord, HandoffPacket, and FencingToken.
- **Model and provider domain:** Isolates vendor specifics through ModelCapabilityRequest, CanonicalModelRequest, and ProviderUsageRecord.
- **Tool domain:** Secures integrations via ToolDefinition, ToolIntent, ToolAuthorisation, and ToolReceipt.
- **Context and memory domain:** Structures operational data via ContextManifest, MemoryItem, ProvenanceRecord, and ContradictionCase.

- **Git and software-delivery domain:** Formalizes CI/CD integration through RepositoryManifest, ChangeManifest, CommitEvidence, and ReleaseEvidence.
- **Governance domain:** Enforces compliance through DecisionRecord, ApprovalRequest, ApprovalDecision, and SecurityControl.
- **Evaluation and quality domain:** Standardizes testing via EvaluationDataset, EvaluationRun, CriterionEvaluation, and CostRecord.
- **Incident and recovery domain:** Tracks failures via IncidentRecord, SecurityFinding, and RootCauseRecord.
- **Artefact and provenance domain:** Guarantees software supply chain integrity via ArtefactManifest, ArtefactReference, ContentDigest, and SBOMReference.
- **Telemetry domain:** Standardizes observability via TraceIdentity, SpanLink, AuditEvent, and CorrelationRecord.

7. Minimum Entity Specifications

The research establishes comprehensive implementation-grade schema structures for the most critical entities. These schemas enforce the separation of durable state from ephemeral provider nuances.

7.1 AgentTask

The AgentTask represents the authoritative state of work. The entity explicitly defines immutability for routing and correlation fields, while execution statuses are strictly managed by policy gateways.

- **Immutable fields:** task_id, task_type, trace_id, correlation_id, idempotency_key, project_id, created_at.
- **Stateful fields:** status, assigned_role, assigned_agent_instance, budget, deadline, retry_policy.
- **Execution parameters:** objective, acceptance_criteria (array of discrete rules), input_artefact_refs (preventing massive payload bloat), required_output_schema, security_classification.

7.2 CompletionState

The architecture strictly rejects reducing task completion to a simple boolean. Completion requires an agent to supply evidence against each criterion. The lifecycle follows a typed, deterministic Directed Acyclic Graph (DAG).

- **Valid States:** CREATED, OFFERED, ACCEPTED, ASSIGNED, QUEUED, RUNNING, PARTIALLY_COMPLETED, WAITING_FOR_INPUT, WAITING_FOR_APPROVAL, VALIDATING, COMPLETED, BLOCKED, ESCALATED, REJECTED, FAILED, CANCELLED, TIMED_OUT, BUDGET_EXHAUSTED.
- **Transition Authority:** Agents cannot transition a task directly from RUNNING to COMPLETED. They must transition to VALIDATING, forcing a deterministic gateway or independent evaluator to compute the criterion-by-criterion results before authorizing the terminal COMPLETED state.

7.3 AgentCheckpoint

To prevent vendor lock-in and ensure resilience against worker crashes, the AgentCheckpoint must cleanly serialize an agent's operational state independently of its underlying provider.

- **Required Fields:** checkpoint_id, task_id, attempt_id, checkpoint_sequence, task_state_version, completed_actions, remaining_actions, context_manifest_id, tool_receipt_ids, decisions, assumptions, budget_consumed, and an integrity_digest.
- **Prohibitions:** The checkpoint explicitly prohibits the inclusion of unrestricted chain-of-thought tokens, provider conversation IDs, or ephemeral process memory. Tying a workflow irretrievably to a specific model's context window prevents a different model family from resuming the state. Assumptions must be explicitly materialized into the assumptions array to ensure resumption logic is fully portable.

7.4 AgentExecutionRecord

This entity represents the complete operational history of a single task attempt, primarily utilized for auditability and EU AI Act compliance.[15] It aggregates the agent identity, provider, adapter version, prompt version, tool calls, costs, state transitions, test execution, final status, and exact Git state during execution.

7.5 HandoffPacket

A HandoffPacket defines the secure transfer of task ownership. When an agent reaches a complexity boundary, it initiates an escalation, delegation, or provider replacement.

- **Fields:** handoff_id, protocol_version, handoff_type, task_id, from_attempt_id, ownership_transfer (boolean), current_state, completed_work, context_manifest_id, checkpoint_id, permissions, and budget_remaining.
- **Continuation Contract:** A handoff is considered complete only when the receiver explicitly acknowledges schema compatibility, task identity, available context, required permissions, and checkpoint readability. The handoff must permit continuation without requiring access to the original provider's API session.

7.6 ToolIntent and ToolReceipt

The architectural separation between ToolIntent and ToolReceipt forms the core security boundary of the tool protocol.

- **ToolIntent:** The model adapter transforms native LLM tool-call output into a canonical ToolIntent, which includes tool_intent_id, operation, arguments, expected_effect, side_effect_class, and idempotency_key. The intent is merely a proposal; a deterministic engine must evaluate it against organization policy and budget.
- **ToolReceipt:** If the intent is authorized, the execution adapter runs the tool and returns a ToolReceipt. This receipt records what *actually* happened, incorporating execution_started_at, execution_finished_at, actual_arguments_digest, result_status, output_artefact_refs, cost, and integrity_evidence. This protects the system against hallucinated state changes and provides an irrefutable audit trail.

7.7 ContextManifest

Context is treated as a compiled, budgeted artifact rather than a raw string of text. The ContextManifest catalogs token_budget, compiler_version, and a collection of ContextItem objects. Each ContextItem carries critical metadata: source, version, provenance, trust_classification, selection_reasons, and valid_time. If context is truncated to fit a smaller model, the manifest explicitly records truncation_actions.

7.8 MemoryItem and DecisionRecord

- **MemoryItem:** Canonical memory resides in structured records encompassing scope, content, provenance, confidence, verification_status, trust_classification, and contradiction_ids. Embeddings and vector indexes are classified strictly as derived data (derived_embedding_refs). Canonical records must be reconstructible without requiring proprietary vector spaces.
- **DecisionRecord:** Tracks autonomous choices, documenting the owner, decision, alternatives, evidence, assumptions, and consequences. This satisfies ISO 42001 explainability requirements by mapping exactly why an agent selected a specific path.[17]

7.9 ArtefactManifest

Large outputs must never be embedded directly in task context payloads, which exhausts token limits and creates brittle database bloat. The ArtefactManifest ensures robust supply-chain integrity, encompassing digest (e.g., SHA-256), media_type, storage_locations, creator_task_id, source_commit_ids, signature_refs, sbom_refs, and attestation_refs.

7.10 Git and Software-Delivery Records

- **ChangeManifest:** Specifies expected repository alterations, including base_commit, allowed_paths, prohibited_paths, and security_sensitive_paths.
- **RepositoryPolicy:** Defines rules for worktree isolation, allowed commands, dependency alterations, and exact-commit test binding. Canonical records must reference exact, immutable Git commit and tree hashes; relying on mutable branch names as release evidence is explicitly prohibited.

8. Canonical Envelope Architecture

To streamline routing, observability, and security filtering, the platform utilizes a unified JSON object envelope for all core protocol records. This ensures that message brokers and tracing gateways can process metadata without parsing complex, nested payload semantics.

JSON

```
{
  "protocol": {
```

```
  "name": "agentic-core",
  "version": "1.2.0",
  "schema_id": "urn:agent-platform:envelope:v1"
 },
 "message": {
  "message_id": "msg_01HGW...",
  "message_type": "task.handoff.proposed",
  "created_at": "2026-07-26T14:57:00Z"
 },
 "identity": {
  "organisation_id": "org_77xx",
  "project_id": "prj_99yy",
  "source_identity": "agent_role_researcher",
  "destination_identity": "agent_role_reviewer"
 },
 "lineage": {
  "trace_id": "4bf92f3577b34da6a3ce929d0e0e4736",
  "span_id": "00f067aa0ba902b7",
  "correlation_id": "cor_1122",
  "causation_id": "msg_01HGV...",
  "task_id": "tsk_8833",
  "attempt_id": "atm_1"
 },
 "delivery": {
  "idempotency_key": "idk_999000",
  "sequence": 14,
  "deadline": "2026-07-26T15:57:00Z",
  "retry_policy": "exponential_backoff"
 },
 "security": {
  "classification": "confidential",
  "trust_level": "verified_internal",
  "required_permissions": ["repo:write"],
  "signature": "sig_..."
 },
 "payload": {
  "type": "HandoffPacket",
  "schema_id": "urn:agent-platform:handoff:v2",
  "data_or_reference": { "data": {... } }
 }
}
```


Design Considerations:

- **Trace Propagation:** Embedding the W3C trace_id and span_id natively within the envelope lineage guarantees that trace context travels alongside the work payload.[1] This overcomes transport-layer limitations where HTTP headers or queue metadata might be stripped by intermediaries.
- **Payload Limits and Nesting:** Envelopes enforce strict size limits (e.g., 1 MB). If a payload exceeds this limit, data_or_reference mandates an ArtefactReference rather than inline data. To prevent recursive schema denial-of-service (DoS) attacks, the schema registry limits payload nesting depth.

9. Identifier and Lineage Standard

The system rigorously differentiates identifiers to preserve logical boundaries and audit trails. Collapsing correlation IDs and task IDs into a single database primary key prevents multi-attempt debugging and violates distributed tracing conventions.

Identifier Type	Generator	Uniqueness Scope	Immutability	Lifetime	Visibility & Rules
Trace ID	W3C Tracer	Global (Mesh)	Immutable	Request Lifecycle	16-byte random hex, generated at ingress, propagated universally.[1]
Span ID	W3C Tracer	Local (Service)	Mutable	Operation	8-byte random hex, generated per service operation.[1]
Task ID	Orchestrator	Project	Immutable	Permanent	Identifies the abstract unit of work, invariant across retries.

Attempt ID	Orchestrator	Task	Immutable	Transient	Identifies a specific, isolated execution try of a task.
Correlation ID	Application	Cross-System	Immutable	Business Logic	Ties disparate asynchronous events to a single originating user action.
Causation ID	Event Bus	Global	Immutable	Permanent	Points to the message_id of the event that triggered the current action.
Idempotency Key	Client	Destination	Immutable	24–72 Hours	Guards against duplicate commands; strictly validated at ingress.
Artefact Digest	Content Hash	Global	Immutable	Permanent	SHA-256 cryptographic hash of an artefact's binary content.

Fencing Token	Lock Mgr	Resource	Immutable	Lease duration	Monotonically increasing integer preventing split-brain writes.

10. Handoff Protocol and Workflow Portability

10.1 Handoff Protocols

Distinct handoff types require precise semantics:

- **Ownership Transfer:** Full accountability shifts to the receiver.
- **Delegation:** A subtask is assigned to a specialist agent, but the parent agent retains ultimate accountability and waits for a Result.
- **Escalation:** Ownership transfers to a higher authority (e.g., human operator or policy engine) due to risk, failure, or budget exhaustion.
- **Provider Replacement:** Work is shifted from one model inference provider to another (e.g., due to an outage) while the agent instance remains constant.

For every handoff, failure behavior, timeouts, retries, and idempotency are strictly governed. A handoff attempt that times out before acknowledgement reverts ownership to the sender for reassignment.

10.2 Workflow Portability Matrix

While portable workflow definitions describe dependencies and retries, claiming complete workflow-engine portability is architecturally naive.

Feature	Portable?	Limitations and Conditions
Task graph	Usually	Depends on a canonical DAG dependency model mapping.
Retry policy	Usually	Requires normalized retry semantics (e.g., jitter, max intervals).
Timer / Wait	Partially	Exposed to clock drift and differing durability

		mechanisms across engines.
Human wait	Usually	Demands external durable state separate from the workflow engine loop.
Child workflow	Partially	Subject to engine-specific lifecycle and termination cascade rules.
Signal / Query	Partially	Requires a canonical message-mapping facade over proprietary read models.
Deterministic replay	Difficult	Irreducibly engine-specific execution constraints (e.g., Temporal's strict non-blocking code rules).
Compensation	Usually	Relies on explicitly mapped saga patterns rather than native engine rollbacks.
Workflow upgrade	Difficult	Version semantics (e.g., continuing inflight vs restarting) vary wildly.

The architecture explicitly maps portable elements (graphs, retries) while isolating engine-specific features (replay determinism, activity heartbeats) behind adapter facades.

11. Checkpoint and Memory Portability Contracts

11.1 Checkpoint Compatibility Levels (C0–C5)

To ensure continuous operation without vendor lock-in, checkpoints must cleanly serialize an agent's operational state. The architecture defines strict Checkpoint Compatibility Levels:

- **C0 (Same process only):** Ephemeral execution state; strictly prohibited for durable orchestration.

- **C1 (Same agent runtime):** Endures standard process restarts and local memory wipes.
- **C2 (Same provider, different instance):** Facilitates horizontal scaling across worker nodes.
- **C3 (Different provider, same platform version):** The minimum baseline for true provider independence. Agent A (using Provider X) checkpoints state; Agent B (using Provider Y) reads the checkpoint and resumes seamlessly.
- **C4 (Different provider and adapter version):** Resilient to forward-compatible schema evolution and minor capability downgrades.
- **C5 (Cross-platform export/import):** The canonical data model is fully serialized and exported to an external, cold-storage ledger for auditing or complete platform migration.

Conformance tests mandate validation for same-provider resume, cross-provider resume, context truncation handling, and expired credential recovery.

11.2 Memory Export and Import Rules

Memory portability demands that canonical memory records (MemoryItem, ProvenanceRecord) are exported with full version histories, access scopes, and contradiction links. Derived data—such as vector embeddings, search indexes, and generated summaries—may be exported separately, but must never be required to reconstruct the canonical truth. Importing memory across platforms involves strict ID collision resolution, provenance verification, and the demotion of unsupported trust classifications to ensure malicious records are quarantined.

12. Provider and Tool Protocol Boundaries

12.1 Provider Adapter Contract

Provider adapters map canonical platform requests into proprietary REST/WebSocket formats. The standard adapter interface implements: discover_capabilities, validate_request, estimate_request, submit_request, stream_events, cancel_request, normalise_response, normalise_usage, and normalise_error.

Capability negotiation determines support for structured outputs, tool calling, vision/audio, reasoning effort, prompt caching, and batch processing. Crucially, adapter conformance is verified via semantic behavioral tests, not merely function signature matching. Two different provider adapters processing the same CanonicalModelRequest must yield semantically equivalent CanonicalModelResponse and ProviderUsageRecord outputs.

12.2 Tool Protocol Boundary

Tools interact via a heavily mediated sequence:Agent/Model → proposes ToolIntent → schema validation → policy evaluation → approval evaluation → transport adapter (MCP/OpenAPI/CLI) → tool execution → ToolReceipt → artefact/structured result → task state update.

This boundary guarantees that external transports (like MCP or OpenAPI) generate a canonical ToolReceipt. Tool transports are strictly forbidden from defining task completion or workflow authority.

13. Error Taxonomy, Retries, and Idempotency

13.1 Canonical Error Model

Provider-native error codes (e.g., AWS JSON exceptions, OpenAI validation errors) are intercepted at the adapter layer and mapped to a canonical error taxonomy.

- **Error Families:** transport, authentication, authorisation, rate_limit, timeout, provider_unavailable, unsupported_capability, invalid_schema, invalid_state_transition, budget_exceeded, policy_violation, tool_failure, integrity_failure.
- **Required Fields:** error_id, error_code, error_family, message, retryability, retry_after, severity, source_component, related_record_id, and occurred_at.
- **Security:** Sensitive provider nuances, underlying stack traces, or internal IP addresses are strictly redacted in cross-boundary error propagation. safe_details are exposed, while sensitive_details_ref points to securely vaulted logs.

13.2 Delivery and Idempotency

Semantics for at-most-once, at-least-once, and effectively-once delivery are defined per primitive. Non-idempotent external actions must never be retried blindly.

- **Commands:** Require mandatory idempotency_key enforcement. Duplicate commands yield the cached ToolReceipt of the initial successful execution.
- **Events:** Rely on partition ordering and replayability. Consumers are responsible for deduplication using the event id.
- **Tasks:** Retries are governed by canonical RetryPolicy structures (e.g., exponential backoff) evaluated against budget and deadline limits.

14. Versioning, Extension, and Compatibility Policies

14.1 Version Layers and Compatibility Dimensions

The architecture distinguishes between protocol versions, envelope versions, entity schema versions, provider-adapter versions, and generated SDK versions. Additive schema changes (adding optional fields) are supported via tolerant reader patterns. However, breaking changes (removing fields, enum removals, changed state semantics) require major protocol version increments. The platform employs translation gateways and adapter migrations to ensure dual read/write capabilities during transition windows.

14.2 Extension Policy

Provider extensions are isolated in controlled namespaces (e.g., extensions: { "provider.azure": {... } }). Extension namespaces require strict schema definitions, maximum size limits, and defined stripping rules. Provider extensions are explicitly prohibited from dictating core workflow transitions. If an extension field is stripped during a cross-provider handoff, the task must degrade gracefully without catastrophic failure.

14.3 Schema Registry and SDK Strategy

The authoritative source of truth is the JSON Schema Registry. The pipeline mandates generation direction: Canonical JSON Schemas → generate Python bindings → generate TypeScript bindings → generate OpenAPI/AsyncAPI components → generate test fixtures. Relying on language-specific models (like Pydantic or Zod) as the primary source of truth is rejected, as they inevitably diverge. Generated bindings are subject to rigorous cross-language round-trip tests to trap divergent nullability, numeric precision, and discriminator behaviors.

15. Security, Identity, and EU AI Act Governance

15.1 Threat Modelling and Controls

The protocol layer isolates critical agentic threats: capability spoofing, downgrade attacks, recursive schema denial of service, forged tool receipts, cross-project ID substitution, and untrusted context losing its label.

- **Controls:** Boundary validation, authenticated OIDC workload identities, signed high-risk records, payload limits, and fail-closed unknown version handling.

15.2 Untrusted-Data Propagation

A primary attack vector involves prompt-injected websites. The architecture dictates that any ingested context tagged with trust_classification: untrusted must retain this metadata across every envelope, HandoffPacket, and ToolIntent. Policy gateways automatically reject ToolIntents requiring high permissions if their context provenance traces back to untrusted sources, thereby neutralizing automated privilege escalation.

15.3 AI Logging and Regulatory Governance

To satisfy EU AI Act compliance (Article 12 and Article 19) for high-risk agentic systems, logging is entirely automated and cryptographically sealed.[15]

- Every state transition, approval decision, and tool execution generates an AuditEvent mapped to ISO 42001 governance controls.[17]
- Logs are retained immutably for a mandated minimum of six months (Article 26) to facilitate post-market monitoring and risk identification.[15]
- High-impact schema changes demand impact analyses, compatibility tests, and formalized security reviews. Agents may propose schema optimizations via a Task, but are explicitly barred from silently altering stable registry contracts.

16. Conformance Architecture and Required Experiments

The research demands functioning prototypes and 18 distinct conformance experiments to validate the architecture, moving beyond theoretical design into implementation-grade verification.

Key Experiments Include:

- **Experiment 1 — Canonical task lifecycle:** Implementing task creation, assignment, execution, checkpoint, validation, and completion using exclusively canonical schemas. Success requires proving that no provider conversation is needed to reconstruct state.
- **Experiment 3 — Cross-provider checkpoint resume:** Starting a task on Provider A, halting at a checkpoint, and resuming on Provider B. Success is measured by continuation without repeating work, losing context, or corrupting artefact integrity.
- **Experiment 4 — Two tool transports:** Executing a canonical ToolIntent via an MCP server and an OpenAPI adapter. Both must produce structurally equivalent ToolReceipt records, verifying the transport-agnostic security boundary.
- **Experiment 8 — Generated bindings:** Generating Python and TypeScript bindings and performing full round-trip serialization and validation for every core entity to trap cross-language numeric or nullability discrepancies.
- **Experiment 10 — Untrusted-context propagation:** Injecting malicious content into a document, proving the untrusted metadata survives every envelope transition and successfully triggers the policy gateway to block unauthorized tool authority.
- **Experiment 15 — Trace propagation:** Tracing a workflow from task initiation, through provider model calls and tool executions, into Git operations and release evidence, verifying flawless W3C and OpenTelemetry correlation.[1]
- **Experiment 16 — Git provenance:** Binding code changes and test reviews to exact Git commit hashes. Pushing a new commit must deterministically invalidate the previous commit's stale approval evidence.
- **Experiment 18 — Protocol fuzzing:** Subjecting the endpoints to missing fields, invalid enums, oversized payloads, recursive data, and malicious extensions to verify safe, fail-closed degradation.

17. Phased Implementation Roadmap

To manage operational burden, the architecture is deployed across a sequential roadmap:

- **Phase 0 (Terminology and primitives):** Establish the schema registry, define communication primitives, select the JSON Schema dialect, and strictly prohibit natural-language coordination for state.
- **Phase 1 (Core task protocol):** Implement AgentTask, CompletionState, acceptance criteria, canonical error models, and W3C trace identifiers.[1]
- **Phase 2 (Execution continuity):** Deploy AgentExecutionRecord, AgentCheckpoint, and HandoffPacket, enabling C3 cross-provider resume capabilities.
- **Phase 3 (Tool mediation):** Establish the ToolIntent / ToolReceipt division and deploy secure MCP/OpenAPI transport adapters.
- **Phase 4 (Context and memory):** Implement ContextManifest and MemoryItem, enforcing trust classification propagation.
- **Phase 5 (Git and delivery):** Roll out ChangeManifest and RepositoryPolicy, binding exact Git evidence and eliminating mutable branch reliance.
- **Phase 6 (Events and telemetry):** Integrate CloudEvents mapping, AsyncAPI generation,

OpenTelemetry GenAI conventions, and implement EU AI Act compliant six-month log vaults.[3]

- **Phase 7 (Identity):** Enforce OAuth/OIDC workload identities and external A2A-style gateways.
- **Phase 8 (Artefacts):** Mandate content-addressed artefacts, OCI integration, SBOMs, and in-toto signatures.[2]
- **Phase 9 (Federation):** Enable cross-organization compatibility, version negotiation, and strict protocol governance.

18. Anti-Patterns to Reject

The integrity of the orchestration platform is maintained by explicitly rejecting pervasive industry anti-patterns:

1. **Unrestricted agent-to-agent chat for coordination:** Causes catastrophic fragility; parsing errors lead to invalid state transitions. *Safer alternative:* Strict JSON Schema validation for all transitions.
2. **Using provider conversation history as task state:** Locks the workflow into a single vendor's specific context window constraints. *Safer alternative:* Provider-independent AgentCheckpoint tracking completed criteria.
3. **Using MCP as the complete orchestration protocol:** MCP lacks durable task coordination, workflow state, and retry semantics.[14] *Safer alternative:* Bounding MCP strictly to the agent-to-tool integration layer.
4. **Mixing commands and events:** Treating an immutable historical event as a trigger for a state mutation creates race conditions and idempotent failures. *Safer alternative:* Strictly separating Command (imperative) and Event (historical) primitives.
5. **Embedding large artefacts in task payloads:** Exhausts token limits and creates massive, unmanageable database rows. *Safer alternative:* OCI Artefact references using SHA-256 digests.
6. **Unversioned schemas and unknown fields:** Permits silent failures and data corruption. *Safer alternative:* Strict $schema validation and additionalProperties: false in core envelopes.
7. **Treating embeddings as canonical memory:** Breaks memory portability when embedding models change. *Safer alternative:* Storing memory in structured, exportable relational formats.
8. **Blindly retrying non-idempotent actions:** Leads to duplicate side effects and data corruption. *Safer alternative:* Mandatory idempotency_key enforcement at the gateway layer.

19. Final Synthesis

This research culminates in a definitive set of architectural mandates that answer the primary research questions.

What is the canonical unit of work? The AgentTask, defined by a rigid objective, explicit acceptance criteria, and bounded operational scope.

What is the authoritative representation of task state? The CompletionState lifecycle, supported by a CriterionResult array detailing verified, evidence-backed progress.
Which communication primitives are required? A strict taxonomy of 20 primitives, notably segregating Task (stateful work), Command (imperative mutation), Event (historical fact), and Handoff (ownership transfer).
Which records may contain natural language, and which must not? Natural language is restricted to objectives, rationale, human-readable summaries, and unstructured tool outputs. It must never represent task identities, execution statuses, budgets, security classifications, approval requirements, or protocol versions.
Which protocols govern interoperability?

- *Model-provider calls:* An internal normalized adapter interface handling capability negotiation, masking specific REST/WebSocket transports.
- *Tools:* The ToolIntent / ToolReceipt standard, wrapping underlying transports (e.g., MCP, OpenAPI).
- *Events:* CloudEvents version 1.0.2.[6]
- *Agent Coordination:* Controlled gateways mapping external A2A structures into the internal AgentTask schema.

What are the boundaries of workflow and checkpoint portability? Task graphs, dependencies, and retries are portable; deterministic replay and activity heartbeats remain engine-specific. A portable checkpoint must contain task versions, completed actions, pending assumptions, and context references. It must explicitly exclude provider conversation IDs, hidden LLM reasoning states, and process-specific memory pointers.
What is canonical memory, and what is derived? Canonical memory consists of structured, versioned records of facts, decisions, and assumptions complete with cryptographic provenance. Vector embeddings, search indexes, and summaries are derived, reproducible data.
How is identity, authority, and lineage represented? Agent identity represents a logical role or instance; provider identity refers to the transport hardware. Authority is represented via cryptographically signed ApprovalDecision records referencing specific ApprovalRequests. Across all operations, the W3C trace_id and the originating correlation_id must propagate flawlessly.[1]
How is artefact and Git provenance preserved? Artefacts use OCI standards when representing complete software supply chain outputs, and ordinary content-addressed storage for standard files. Git provenance is linked exclusively via absolute, immutable commit hashes and tree hashes.
How are evolution and governance managed? Core schemas serve as stable integration contracts. Provider extensions utilize tightly scoped, isolated namespaces. Enum evolution is closed for state transitions and open for diagnostics. Removing fields or changing state semantics requires a major protocol version. When version negotiation fails, the system safely fails closed. Schema validation failures block CI, and conformance failures block releases.
What evidence would invalidate the final recommendation? If major foundation models completely eliminate the necessity for API calls by embedding infinitely durable, standardized state machines directly into localized hardware—rendering distributed orchestration

redundant—the requirement for this complex distributed protocol abstraction would be invalidated. Until such a paradigm shift occurs, this architecture provides the most robust, provider-independent foundation for enterprise agentic systems.

Works cited

1. W3C Trace Context Explained: Traceparent & Tracestate - Dash0, accessed July 26, 2026, https://www.dash0.com/knowledge/w3c-trace-context-traceparent-tracestate
2. attestation/spec/v1/statement.md at main · in-toto/attestation - GitHub, accessed July 26, 2026, https://github.com/in-toto/attestation/blob/main/spec/v1/statement.md
3. AsyncAPI Initiative for event-driven APIs | AsyncAPI Initiative for event-driven APIs, accessed July 26, 2026, https://www.asyncapi.com/
4. Releases · asyncapi/spec - GitHub, accessed July 26, 2026, https://github.com/asyncapi/spec/releases
5. Cloud Events - GitHub Pages, accessed July 26, 2026, https://tedneward.github.io/Research/distribution/cloud-events.html
6. Releases · cloudevents/spec - GitHub, accessed July 26, 2026, https://github.com/cloudevents/spec/releases
7. NL GOV profile for CloudEvents 1.1 - Logius, accessed July 26, 2026, https://gitdocumentatie.logius.nl/publicatie/notificatieservices/cloudevents-nl/
8. Trace Context - W3C, accessed July 26, 2026, https://www.w3.org/TR/trace-context/
9. Trace Context Level 2 - W3C, accessed July 26, 2026, https://www.w3.org/TR/trace-context-2/
10. How OpenTelemetry Traces LLM Calls, Agent Reasoning, and MCP Tools | Greptime, accessed July 26, 2026, https://greptime.com/blogs/2026-05-09-opentelemetry-genai-semantic-conventions
11. OpenTelemetry GenAI Semantic Conventions | MLflow AI Platform, accessed July 26, 2026, https://mlflow.org/docs/latest/genai/tracing/opentelemetry/genai-semconv/
12. Inside the LLM Call: GenAI Observability with OpenTelemetry, accessed July 26, 2026, https://opentelemetry.io/blog/2026/genai-observability/
13. How to Monitor LLM Applications with OpenTelemetry GenAI Semantic Conventions, accessed July 26, 2026, https://oneuptime.com/blog/post/2026-02-06-monitor-llm-opentelemetry-genai-semantic-conventions/view
14. Specification - Model Context Protocol, accessed July 26, 2026, https://modelcontextprotocol.io/specification/2025-03-26
15. AI Act Record-Keeping: What High-Risk Systems Must Log - TrueScreen, accessed July 26, 2026, https://truescreen.io/insights/ai-act-record-keeping-requirements/
16. AI Logging Under the EU AI Act: The Compliance Infrastructure Behind High-Risk

Systems, accessed July 26, 2026, https://www.dsn-group.com/privacy-notes/ai-logging-under-the-eu-ai-act-the-compliance-infrastructure-behind-high-risk-systems-4458904
17. Is Your AI Logging Audit-Proof? Unlock Article 19 Compliance With ISO 42001 Controls, accessed July 26, 2026, https://www.isms.online/iso-42001/eu-ai-act/article-19/