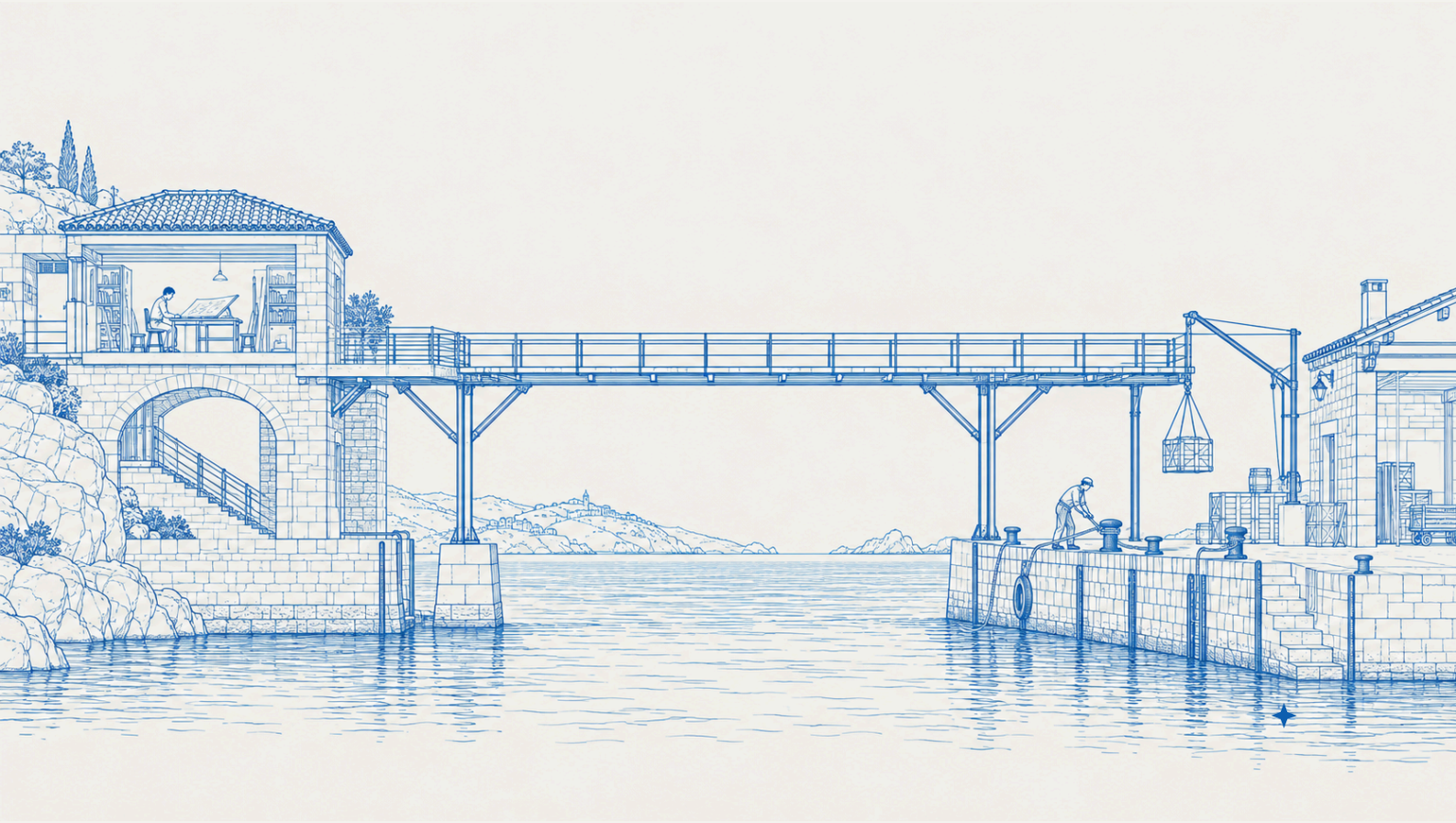
GOVERNED INTELLIGENCE · WHITEPAPER

Designing Agentic AI for the Enterprise

An architecture for knowledge, decisions and reliable execution

Timotheos Samartzidis

October 30, 2026 · English edition · 4.1



From a convincing proposal to reliable action

An AI agent can read a report, recognize connections and suggest a next step. As soon as that suggestion is meant to change something in a business, some familiar questions become relevant: Which version of the data applies? Which rule should be used? Who is responsible? And who is actually allowed to initiate the change?

The agent may answer these questions in impeccable English. The system of record is generally unmoved by its eloquence. It needs valid inputs, appropriate permissions and a precisely defined action. This is where enterprise architecture—the design of the relationships between business, data and IT—becomes concrete for agentic AI. Here, an AI agent combines a language model with tools and software capable of carrying out several steps. The language model, often called an LLM, short for large language model, interprets the request. The surrounding architecture must determine which action may follow.

My recommendation is to design enterprise knowledge, decisions and execution as one connected system. The knowledge base describes the things the business talks about and connects them to their sources. The decision architecture assigns tasks to appropriate mechanisms: code calculates defined values, rules check explicit conditions, and LLMs interpret language and develop proposals. The execution architecture ensures that an authorized action takes effect under the right conditions.

The agent runtime is the software that brings these contributions together while a request is being processed. It maintains the request across steps, retrieves context, assigns work to components and processes their results. This makes it possible to trace how information leads to a decision and how a decision leads to a confirmed action.

If you are responsible for enterprise architecture, data, platforms or business applications, this design should help you structure a use case of your own. Which sources does it need? Which decisions belong in code, rules or human review? And which guarantees must the technical execution provide? The outcome is a scoped architecture brief with named deliverables. The introduction and knowledge model establish the business question; the clearly marked technical details that follow explain the runtime, action contract and resumption of work.

This whitepaper develops that connection from the business knowledge model to an outline of a technical runtime. The components and contracts presented here are an architecture proposal, not a platform that has already been implemented or tested. It starts with a requirement that can seem self-evident: everyone involved needs to know what they are talking about.

Connecting enterprise knowledge to its sources

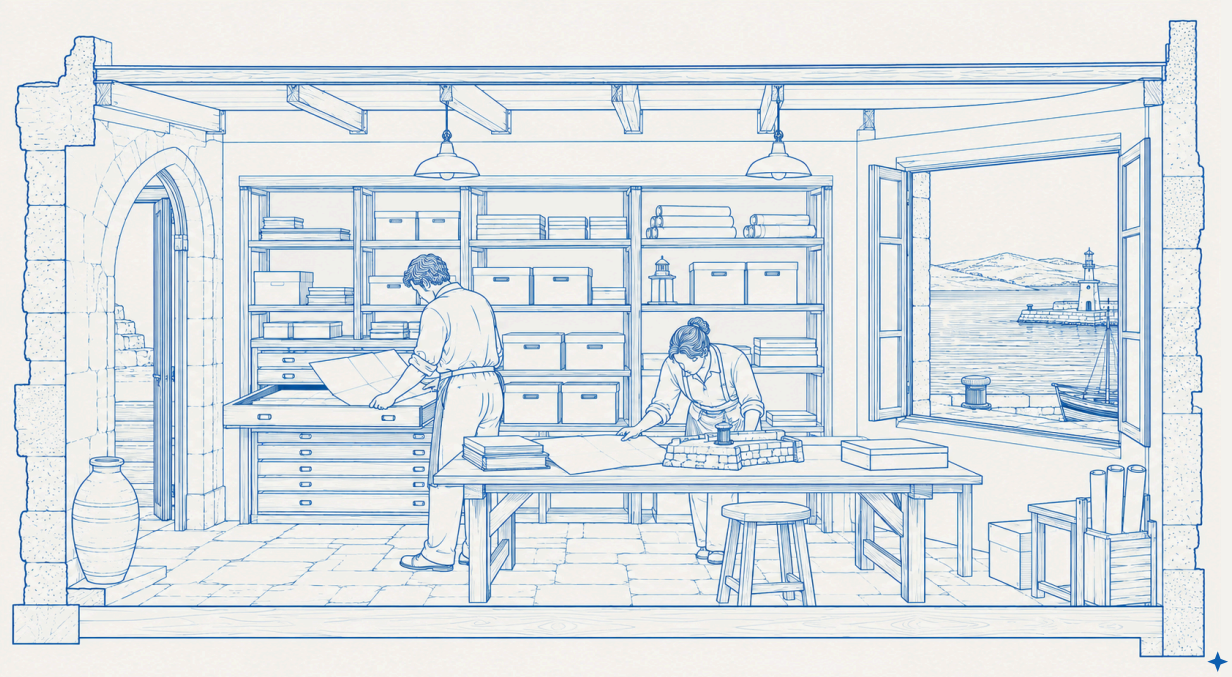
An organization's knowledge is spread across business documents, databases, applications and the people who understand their particular quirks. An agent needs access to these relationships. It must be able to recognize which terms mean the same thing, which data belongs together and which source is authoritative for a particular question. The following building blocks contribute different parts; none replaces the others.

Building block	Contribution to the shared knowledge model
Enterprise architecture	Connect capabilities, processes, applications, data, responsibilities and dependencies.
Wiki and business documents	Retain explanations, reasoning, exceptions and business context.
Glossary and taxonomy	Define terms and organize categories consistently.
Business ontology	Describe business objects, properties and domain relationships in a machine-readable form.
Knowledge graph	Connect concrete objects and relationships to their sources.
Systems of record	Maintain authoritative records and transactions within their respective areas of responsibility.

A glossary explains terms; a taxonomy organizes them into categories. An **ontology** describes business objects and their domain relationships, such as how a contract, a risk and an event relate. A **knowledge graph** records concrete objects and known connections. A contract object can, for example, refer to its documents, the responsible business unit and the authoritative record. The conceptual model and the concrete data remain distinct.

Stable identifiers—references that can be matched across systems—connect these views. A **system of record** holds the authoritative record or performs the authoritative transaction within its area of responsibility. For each request, it must be clear which source is authoritative for which claim. Access to documents and data remains limited to the permitted context; putting them in a graph does not extend anyone's permissions.

This establishes meaning. Whether a statement is valid today still needs to be checked against its sources. Information relevant to a decision therefore needs an owner, a business validity period and a retrieval time as part of its context. Observed facts, estimates, hypotheses and contradictions remain distinguishable. An agent's conjecture acquires a different status only through a review process with accountable ownership.



Sources must be accessible, attributable and accountable. Fictional archive · AI-generated

Different standards are available for implementation. SKOS supports knowledge organization systems; OWL supports formal ontologies. SHACL describes testable constraints on data in the RDF graph model, such as required properties and permitted values. Passing a structural check does not establish business truth. Which of these means you need depends on the use case and your existing systems.

PRACTICE GUIDE · ARCHITECTURE BRIEF

Map the knowledge needed for one decision

Choose a specific decision and capture only the terms, objects and sources it requires. For each item of information, record its business owner, authoritative source, identifier, validity and permitted access. Mark unresolved contradictions. The expected result is a small knowledge model with accountable ownership, source mappings and open questions. A complete model of the entire enterprise is not a prerequisite.

This defines what the work may rely on. The next architecture task is to use that knowledge appropriately for the decision at hand.

Assigning work to code, rules and LLMs

Calculating a deadline, applying a rule and interpreting a contradictory document are different tasks. A sound decision architecture makes those differences explicit. Each component can then be used where its result can be meaningfully checked.

Mechanism	Suitable task	What supports the result
Deterministic code	Calculate amounts, deadlines and defined functions.	Reproducibility given the same inputs, state and implementation.
Rules / DMN (Decision Model and Notation)	Evaluate explicit conditions and decision tables.	Application of a documented rule to a particular set of inputs.
LLM interpretation	Interpret language, identify contradictions and propose next steps.	A reviewable proposal within a bounded assignment.

Decision Model and Notation, or DMN, is a standard for modeling business decisions, including through decision tables. A rule can check an explicitly defined condition. A language model may first identify a matter in a document that needs review. Defined code is suitable for the subsequent calculation. The architecture determines how these contributions fit together and where their results must be confirmed.

An LLM result becomes binding for the business only after the prescribed review and authorization. How convincingly it is written does not settle that question. Even a correctly evaluated rule produces the intended decision only if the rule version and inputs are appropriate.

Alongside the business decision comes the question of whether the specific action is permitted. Here, a **policy** is a machine-evaluable requirement concerning authority and conditions. A **policy engine** uses structured information to decide whether that requirement is met; the executing service enforces the decision. Open Policy Agent separates these responsibilities accordingly.

Consequences matter too. Jeff Bezos distinguishes readily reversible Type 2 decisions from Type 1 decisions that are difficult to reverse. This dimension helps determine the review required. On its own, it does not determine which tool may contribute to preparing a decision.

In the design below, human approvals depend on business responsibility, uncertainty and possible consequences. An approval applies to a specific action under specific conditions. Together with the knowledge model, this leads to the overall architecture.

How knowledge, orchestration and business systems work together

Knowledge, decision mechanisms and executing systems need to refer to one another. Figure 1 organizes their logical responsibilities. Read it from top to bottom: a business request uses the shared knowledge base; orchestration coordinates the work and selects appropriate mechanisms; the specific action is checked before a change takes effect. The business system then confirms the result. Products, network paths and the distribution across individual services remain open.

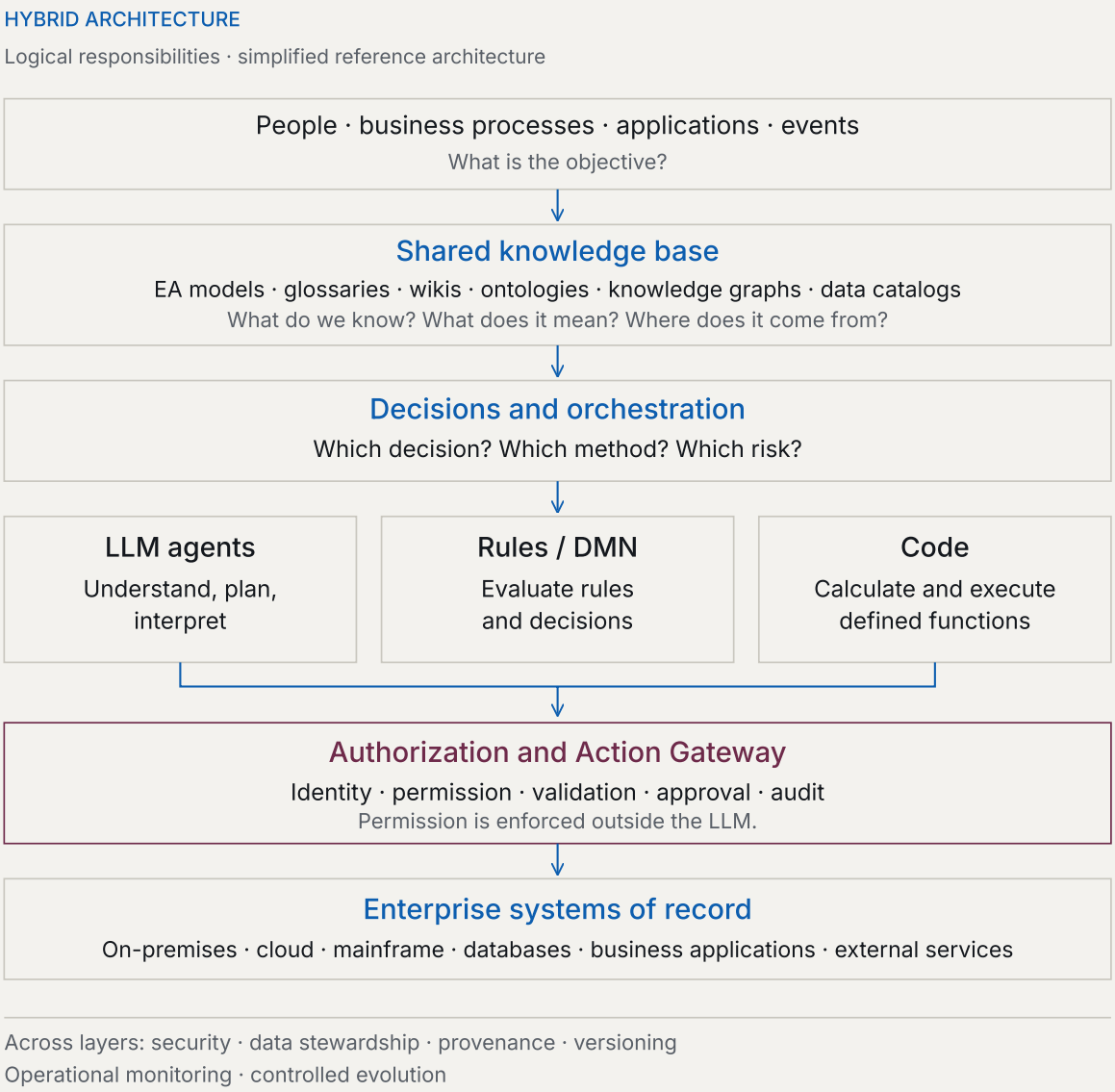


Figure 1. Logical responsibilities from request to business system. Arrows show conceptual relationships; cross-cutting responsibilities apply at every layer. Original reference design.

The arrows show this logical relationship. They are not network connections and do not describe security boundaries that have already been established. The bottom row names responsibilities that apply across all layers: security and accountable data stewardship, information provenance, versioning, operational monitoring and controlled evolution.

Shared meaning connects the layers. When a business rule changes, its impact on the agent's context, rule evaluation and affected actions must be assessed. Enterprise architecture makes those relationships traceable. Business owners are accountable for the content; engineering and operations implement and operate the technical mechanisms.

Related building blocks appear in Palantir's ontology architecture and the AWS guidance on agentic enterprise architecture. Their combination in this illustration is my own reference design. Similarity between individual components does not establish an identical implementation or demonstrate the effectiveness of the overall design. The following runtime sketch explains how the roles are intended to work together during a request.

How the runtime carries a request across steps

TECHNICAL DETAIL

Components and their responsibilities

A request and its result are often separated by several steps, follow-up questions and waiting periods. The runtime must preserve that continuity. Here, a **run** is the processing of one request, with its own identifier and persisted state. It may include many model and tool calls and last longer than any individual call.

The following names identify roles in the design. They prescribe neither products nor a separate service for every box. Existing workflow, integration or application services can take on several of these responsibilities, provided the necessary guarantees remain verifiable.

Role in the design	Responsibility during a run
Run Manager	Verify the caller and bind the request, delegated authority, identifier and limits.
Event queue	Make new work, received results and permitted continuations available for processing.
Orchestrator	Coordinate the next bounded step and control state transitions.
Context Builder	Assemble information permitted for the request, together with its provenance and data version.
Run Store	Persist state, pending actions, approvals and evidence.
Worker	Carry out a bounded assignment using an LLM, rules or code, and return a result.
Action Gateway	Check a specifically proposed change before forwarding it to the business system.

An **Action Intent** describes the intended change, including its target, inputs and execution context. A **tool adapter** translates the authorized request into a call to the relevant application programming interface, or API. Using these roles, Figure 2 shows how a run is intended to reach a confirmed action.

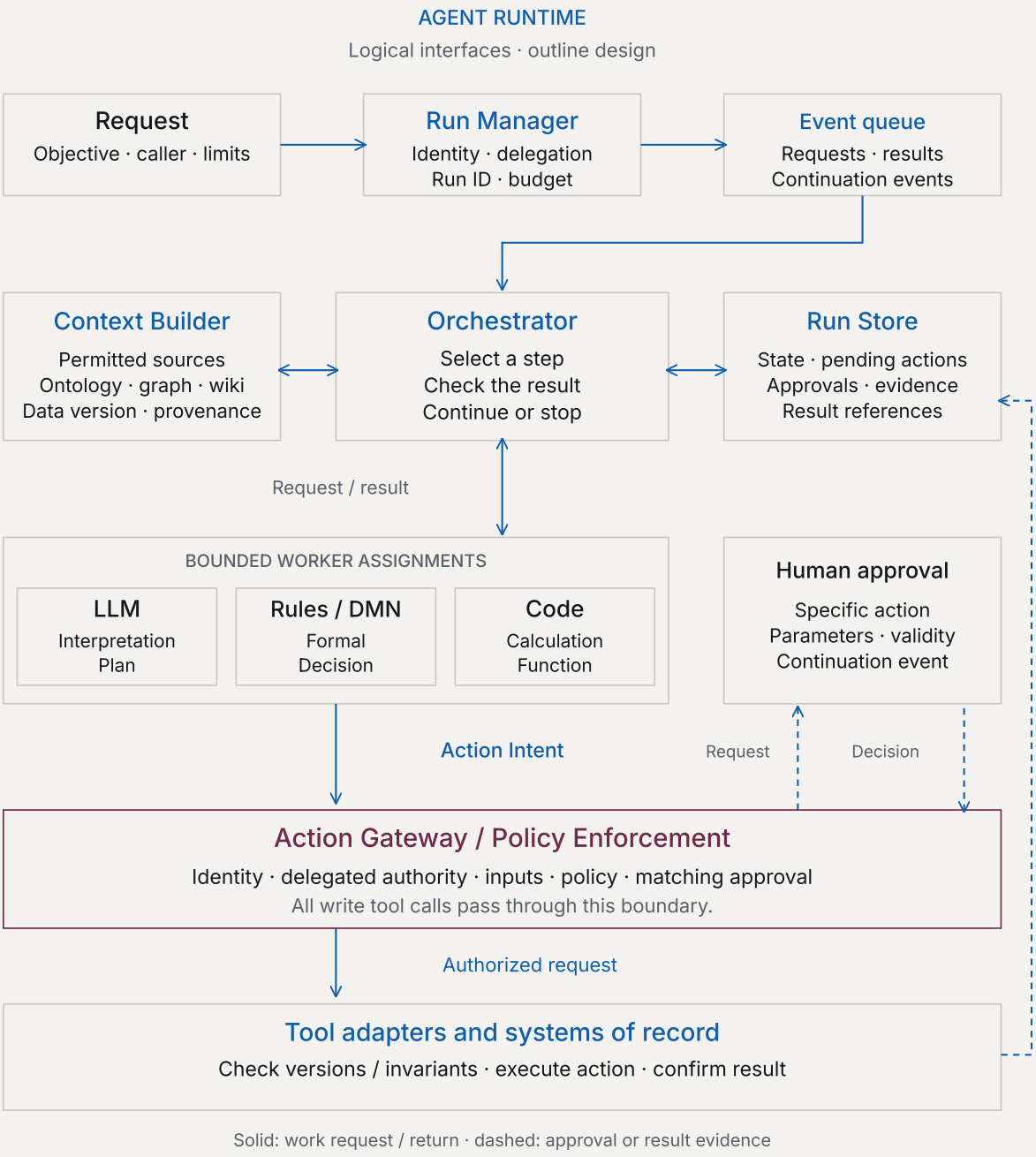


Figure 2. Roles and logical interfaces of the proposed runtime. Solid arrows: work requests or returns. Dashed arrows: approvals or result evidence. Not a validated implementation.

Solid arrows show work requests and returns; dashed arrows show approvals or evidence of results. Context retrieval and persisted processing state are available to the Orchestrator as separate responsibilities. The return path on the right attaches confirmation from the business system to the request. An approval received later can trigger further processing through a continuation event; it does not execute the action itself.

The highlighted gateway boundary is an implementation requirement: tool calls that write data must pass through this check. Workers must therefore have no equivalent write access that bypasses it. How permissions, runtime isolation and interfaces actually enforce this boundary must be demonstrated in the concrete technical design. The sequence within these roles can now be followed step by step.

From a request to the next confirmed step

An agent needs room to find a useful next step. The runtime needs clear conditions for allowing that step. The Orchestrator connects the two in a recurring cycle and controls its transitions outside the LLM.

1. **Bind the request.** The Run Manager authenticates the caller and determines the delegated authority. The objective, budget and stopping conditions share a common reference through the run ID.
2. **Take ownership of the state.** The Orchestrator takes a new or resumable request from the queue. An exclusive processing claim and version-checked state changes are intended to prevent competing updates to the same run. The implementation must also prevent a stale worker from writing further valid changes after losing its claim.
3. **Retrieve context.** The Context Builder reads permitted business objects and sources. Data versions, provenance and open questions accompany the assignment.
4. **Determine the next step.** An LLM, rule component or code returns a structured result. The runtime checks its schema, business conditions and permitted scope.
5. **Execute or wait.** Read-only tools operate in the run's access context. A write action produces an Action Intent. If information or approval is missing, the runtime durably records the waiting state.
6. **Authorize the action.** The gateway checks the specific intent against current authority, rules and any required approval. Changed parameters or relevant data versions require another check.
7. **Confirm the result.** The adapter reports success or failure. If the outcome remains unknown, reconciliation with the target system begins. Only a confirmed result changes the known execution state.
8. **Continue or finish.** The Orchestrator checks the objective and limits. A further step requires remaining budget and a permitted state transition.

The LLM develops its proposals within this framework. The runtime controls step counts, deadlines, costs and termination. As soon as a proposal is to trigger an authoritative change, it needs an unambiguous description. That is the purpose of the action contract.

The action contract specifies the target, inputs and approval

TECHNICAL DETAIL

Understanding the fields of the action contract

“Take the next sensible step” is hardly sufficient as an instruction to a business system. It must be clear which action is to change which object, using which inputs. An Action Intent records these details together with the execution context. The following JSON—JavaScript Object Notation, a structured data format—outlines that contract. It contains illustrative values and placeholders. It is neither a complete interface schema nor an executable implementation.

```
{
  "runId": "run-17",
  "subjectRef": "actor-12",
  "delegationRef": "delegation-4",
  "actionId": "action-7",
  "targetRef": "record-42",
  "expectedVersion": "v19",
  "inputRef": "input-3",
  "inputHash": "sha256:…",
  "policyRevision": "policy-6",
  "approvalRef": null,
  "idempotencyKey": "record-42:action-7",
  "status": "PROPOSED"
}
```

In this example, proposed action `action-7` belongs to run `run-17` and concerns `record-42`. Its inputs are referenced by `input-3`; they are not written out here. `PROPOSED` means the action has been proposed. It does not establish that execution has occurred. The field names bring together the following questions:

Fields	Meaning and responsible check
`runId`	Which request does the action belong to? The runtime associates it with the persisted run.
`subjectRef`, `delegationRef`	Who acts on whose behalf? Trusted identity and runtime services supply the references; the gateway checks the authority.
`actionId`, `targetRef`, `inputRef`	Which specific action concerns which target and inputs? The adapter and gateway must be able to resolve and validate the action type and contents unambiguously.

Fields	Meaning and responsible check
`expectedVersion`, `policyRevision`	Which object and policy versions are the basis? The gateway checks the applicable policy; the target system checks its authoritative object version.
`inputHash`	Which version of the inputs does the check cover? A hash is a check value calculated from the data. Its calculation and binding to the inputs actually executed must be defined.
`approvalRef`	Where is any required approval evidenced? Here, `null` means that no reference has been entered. The applicable policy determines whether execution without approval is allowed.
`idempotencyKey`	Which retries belong to the same logical action? The runtime assigns the key; the executing service must process it. The target system must reliably prevent duplicate business effects.
`status`	Which action state is confirmed? The runtime changes state on the basis of verified events.

In the example, `actionId` assumes an unambiguous mapping to a registered action. A real interface schema must also define how the action type, parameter structure and permitted target objects are determined. The shortened value `sha256:…` is not an actual hash. The idempotency key is also illustrative: different business actions must not accidentally receive the same key.

The agent proposes the action and supplies its business reasoning. Trusted runtime components add the verified caller context, authority references and evidence. A permission the model awards itself has roughly the evidentiary value of a power of attorney it signed for itself.

An approval binds the action, inputs, target and relevant versions. If any of these changes, the earlier approval is no longer sufficient. Immediately before the change takes effect, the target system checks the applicable conditions and versions within the transaction. A prior gateway check alone cannot rule out intervening changes.

The Model Context Protocol, or MCP, describes, among other things, how tools expose input schemas and, where applicable, output schemas. These descriptions help callers use tools correctly. For the action contract proposed here, the participating services must additionally enforce identity, approval binding and a confirmed result.

PRACTICE GUIDE · TECHNICAL ASSIGNMENT

Agree the action contract with the target system's team

Select one action that writes to the business system and work through every field in the example with the application team: who sets it, where is it stored and who checks it? Add the action type, schema, error responses and the binding of an approval. Document how the target system handles versions and repeated calls. The expected result is an agreed interface contract with open implementation questions. The JSON alone does not fulfill this assignment.

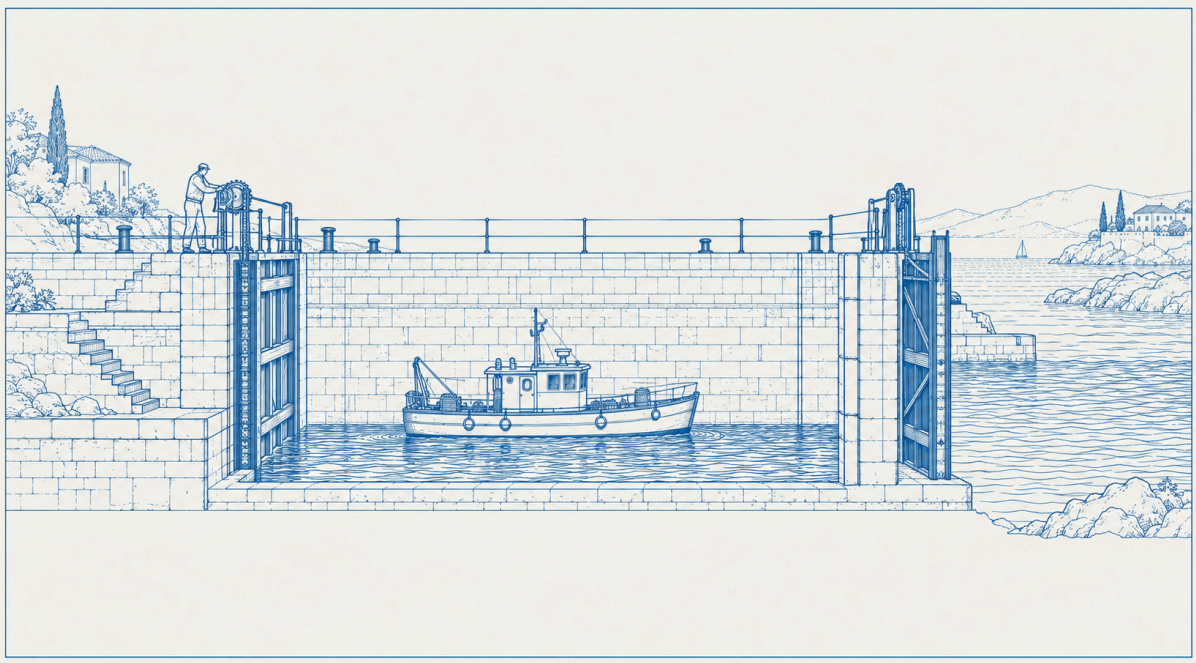
Enforcement therefore remains an architecture responsibility. The time that may pass between proposal and execution leads to the next requirement: the request must survive an interruption.

Persisting requests, reconciling unknown outcomes and resuming work

TECHNICAL DETAIL

Persisted state and resumption

An approval may arrive long after the original model call has ended. A service may have made a change even though its response never arrives. In both situations, the runtime needs persisted state from which a permitted continuation can be determined.



After a wait, the conditions for the next step must still hold. Fictional lock · AI-generated

Run state	Meaning and possible continuation
QUEUED / RUNNING	The request is waiting for processing or carrying out a bounded step.
WAITING_INPUT	Specified information is missing; a matching event may resume the run.
WAITING_APPROVAL	A specific action is waiting for approval from an authorized decision-maker.
RECONCILING	An action's outcome is being established; dependent actions remain blocked.
COMPLETED	The defined objective has been reached with a confirmed result.
STOPPED / FAILED	A stopping condition or error prevents automatic continuation; the reason is retained.

These states are a design proposal for the run. Individual actions have their own progression from proposal through checking and dispatch to confirmation. **UNKNOWN** denotes an unresolved outcome after dispatch. A timeout initially confirms that a response is missing. It is remarkably restrained on the question of whether the action succeeded.

When an approval arrives later, the runtime loads the persisted action and rechecks its validity, unchanged parameters and current decision context. A persisted history makes resumption possible; earlier permissions must still be valid.

A retry of the same logical action should not cause an additional business effect. This property is called **idempotency**. The design uses the same idempotency key for the same action; a new business request receives its own mapping. The executing service must recognize duplicate calls; the target system must reliably prevent additional business effects on repetition. The key alone does not create this property. The Temporal documentation on Activities explains this relationship too. A persisted workflow alone does not guarantee a single business effect in arbitrary external systems.

If an outcome can neither be established nor safely retried, the action remains on hold for review. Stopping a run does not undo a business effect that has already been confirmed. A **compensation**, such as a permissible reversing entry, is a separate action requiring fresh authorization. These limits belong in the runtime design and must be verifiable in operation.

Checking execution through controls and evidence

The architecture's quality becomes visible at its transitions: from request to context, from proposal to approval, and from execution to a confirmed result. For each transition, it must be clear which condition applies and what happens when it is missing.

Control point	Behavior when a condition is missing
Caller identity and delegated authority	Reject a run without a verified caller and bounded authority.
Context and data version	Identify missing, outdated or contradictory information; request clarification or domain review when needed.
Result and tool schemas	Reject unsuitable outputs; formally valid results still require business validation.
Policy and approval	Block unauthorized actions; represent missing approval as a waiting state.
Versions in the target system	Reject changes when conditions are violated or versions conflict.
Budget and termination	Prevent further steps when the budget is exhausted, a deadline is exceeded or a stop decision is made.
Unknown action outcome	Reconcile status and block dependent actions until the outcome is established.

The evidence connects the run to its task, sources and data versions, model and template versions, rules, policies, approvals and tool calls. It records the confirmed result with a reviewable rationale. This requires relevant evidence; the model's internal reasoning is not necessary as an audit record. Access, retention and sensitive content in these records are themselves part of the technical design. Full document contents or secrets do not belong in a log merely because they were available in context.

Responsibilities remain visible too. Business owners maintain terms and rules. Authorized decision-makers are accountable for approvals. Engineering and operations implement the controls and ensure their reliable operation. Enterprise architecture keeps the relationships between these responsibilities traceable as things change.

These control points describe requirements. Whether an implementation meets them must be checked against concrete evidence. The design leaves product selection, interfaces, storage, isolation and performance sizing open. This is the transition from the general architecture to your own implementation brief.

Turning the design into a reviewable implementation brief

To get started, you need a process whose objective and limits your team can name. Choose a bounded decision with a manageable set of sources and a clearly describable possible action. Then develop the knowledge, decision and execution views for exactly that scope. Existing services and responsibilities belong in the discussion from the start; a new box in a diagram is not, on its own, a reason for a new service.

The result should be readable by the business team, architecture, engineering and operations together. The following mapping makes visible which documents may be missing before implementation. The names are suggested filing conventions, not outputs from a tool that has already been run.

Expected artifact	Content and accountable decision
Use case and knowledge model	Objective, limits, terms, authoritative sources, access and business owners. The business team confirms meaning and usable data versions.
Decision map	Assignment of individual steps to code, rules, an LLM and human review, with a rationale and required inputs.
Action and state contract	Trusted origins of fields, approval binding, version checks, retries, waiting states and handling of unknown outcomes. Engineering and the application team agree the interfaces.
Control and operations plan	Evidence for control points, budgets, termination, log access and accountable responses to errors or an unknown action outcome.
Open architecture decisions	Product selection, deployment, isolation, storage and sizing, with a named owner for each.

PRACTICE GUIDE · PROPOSED ACCEPTANCE TESTS

Test approval and resumption under failure conditions

In a disposable test environment, check what happens when delegation is missing, parameters change after approval, versions conflict or a call is duplicated. Also interrupt a request after dispatch but before the response: resumption must establish the outcome and must not allow a dependent action to continue unchecked. Record expectations, observed behavior and evidence for each case. These proposed acceptance tests were not executed for this reference design.

This gives the team a basis for detailed technical design and later acceptance. Before use, operations, protection requirements and performance must also be assessed for the actual system. Passing a test of one transition does not establish the reliability of the entire process. It does, however, make a particular guarantee verifiable and show which questions remain open.

Conclusion

A convincing proposal becomes a reliable action only when meaning, the decision process and execution fit together. The agent needs sources with accountable ownership; each step needs a suitable mechanism; the change in the business system needs specific authority and a confirmed result. Enterprise architecture connects these guarantees across responsibilities and changes.

For your own design, it is therefore worth working through one bounded request from beginning to end. Establish where each item of information comes from, who is accountable for each decision, and under which conditions an action may be executed, retried or stopped. The resulting knowledge models, contracts and test cases give implementation a sound working brief.

The components, states and data fields shown here form a conceptual reference design. Their combination has not been implemented or experimentally validated. Standards and product documentation inform individual building blocks; they do not demonstrate that the architecture as a whole has been tested. The detailed technical design must enforce and demonstrate the described boundaries in the systems actually selected.

The practical yardstick remains the opening question: can the business explain and substantiate why this particular action was permitted under these particular conditions? A sound architecture preserves that connection from request to confirmed effect. When a condition is missing, it must be equally able to stop in a traceable way.

Sources and design status

Sources are linked beside the claims they support. This English edition retains all ten primary sources of the German manuscript. The combination of components, states, data fields and proposed acceptance tests is an original conceptual reference design. It has not been implemented or experimentally validated.

1. SKOS
   https://www.w3.org/TR/skos-reference/
2. OWL
   https://www.w3.org/TR/owl2-overview/
3. SHACL
   https://www.w3.org/TR/shacl/
4. Decision Model and Notation
   https://www.omg.org/spec/DMN/1.5/About-DMN
5. Open Policy Agent
   https://www.openpolicyagent.org/docs
6. Jeff Bezos distinguishes
   https://d1.awsstatic.com/onedam/marketing-channels/website/aws/en_US/whitepapers/approved/executive-insights/2015-letter-to-shareholders.pdf
7. Palantir's ontology architecture
   https://www.palantir.com/docs/foundry/ontology/overview
8. AWS guidance on agentic enterprise architecture
   https://docs.aws.amazon.com/prescriptive-guidance/latest/govern-architect-agentic-ai/enterprise-architecture.html
9. Model Context Protocol
   https://modelcontextprotocol.io/specification/2025-11-25/server/tools
10. Temporal documentation on Activities
    https://docs.temporal.io/activity-definition

English edition of version 4.1 · October 30, 2026. Translated in full and independently reviewed against the German manuscript using the Lektor skill. “Security Testing with AI Agents” informs reader guidance and integrated technical details. AI-generated illustrations show fictional places and explain ideas, not validated effects. Both diagrams remain editable vector graphics. Wordmark from the current website; full provenance in CREDITS.md.