QORTARA CLOUD GOVERNANCE

Policy Authoring

Policy Authoring

Good policies are small, explicit, and testable. A policy describes which agent actions are allowed, which are denied, and under what conditions. At runtime the decision point matches an action against your active policies and returns allow, deny, or verify. Deny always wins.

This guide covers both surfaces:

- **Qortara Governance (open-source sidecar).** Policies live in your repository as a YAML policy profile and are evaluated locally on the agent's tool-call path. This is available today (`pip install qortara-governance-langchain`, Apache-2.0) and is where most of this guide is immediately usable.
- **Qortara Cloud Governance (hosted control plane).** Policies are managed centrally as JSON policy packs through a REST API, with server-side versioning, a tamper-evident history, and cross-organization context that a single local process cannot see. The hosted plane is in pre-launch and has not been deployed. The JSON format and endpoints below describe the pre-launch design.

The two formats share the same decision model: action plus resource pattern plus effect plus conditions. If you author against the sidecar today, the concepts carry over to the hosted plane unchanged.

The Sidecar Profile (available today)

The sidecar reads a YAML profile that you name when you construct a `GovernedAgent`. Rules are evaluated in order against each tool call and its inputs, and an explicit deny always beats an allow.

```yaml
profile: crm-readonly
rules:
  - effect: allow
    when:
      tool.name: fetch_customer_record
  - effect: deny
    when:
      tool.category: external_write
```

You select the profile in code:

```python
from qortara_governance_langchain import GovernedAgent

agent = GovernedAgent(
    tools=[fetch_customer_record, delete_customer_record],
    policy_profile="crm-readonly",
)
```

A denied tool call raises `PolicyDenied` instead of executing, and the exception carries the decision so you can log it. Start narrow, run the agent, watch which calls get denied, then widen the profile as you gain confidence. The rest of this guide describes the full rule grammar, including the richer policy-pack format used by the hosted plane.

Policy Pack Format (hosted, pre-launch)

In Qortara Cloud Governance, policies are JSON objects managed through the REST API. Each policy has a name, a description, and a list of rules. This is the same allow/deny/condition model as the sidecar profile, expressed in JSON and stored centrally.

```json
{
  "name": "policy-name",
  "description": "Human-readable description of what this policy does",
  "rules": [
    {
      "action": "action-type",
      "resource_pattern": "glob/pattern/*",
      "effect": "allow | deny",
      "conditions": { }
    }
  ]
}
```

Rule Structure

Each rule evaluates against an agent's action context.

Field	Type	Description
`action`	string	The action type to match: `read`, `write`, `delete`,

`send_email`, `api_call`, `transfer`, or any custom string. |
`resource_pattern`	string	Glob pattern matching the resource: `customers/*`, `customers/*/pii`, `*` (all), `emails/*@*.com`.
`effect`	string	`allow` or `deny`. Deny takes precedence when multiple rules match.
`conditions`	object (optional)	Additional requirements that must hold for the rule to apply.

Matching is by action type and resource pattern. Patterns are glob-style, so `customers/*/pii` matches `customers/12345/pii`. If a rule has no conditions, it applies whenever the action and resource pattern match.

Condition Operators

Conditions add context-sensitive requirements to a rule. A rule with conditions only takes effect when every condition is satisfied.

```json
{
  "conditions": {
    "require": {"capability": "pii-approved"},
    "unless": {"capability": "admin"},
    "min_trust_score": 0.7,
    "time_window": {"after": "09:00", "before": "17:00", "timezone": "UTC"},
    "max_rate": {"count": 100, "period": "1h"},
    "metadata_match": {"department": "engineering"}
  }
}
```

Operator	What it does
`require`	The agent MUST have this capability for the rule to apply.
`unless`	An agent with this capability is EXEMPT from the rule.
`min_trust_score`	The agent's trust score must be at least this value.
`time_window`	The rule only applies during this time window.
`max_rate`	Rate limit: at most N actions per time period.
`metadata_match`	The agent's metadata must contain these key-value pairs.

Capabilities, trust scores, and cross-organization context are richest on the hosted plane, which can see attestations across tenants. The sidecar evaluates the same operators against the context available locally.

Built-In Policy Templates

These templates are ready to use. Copy one, change the names and patterns to match your environment, and test it before deploying. Each template is shown in the hosted

JSON policy-pack format; the same intent can be expressed in a sidecar profile using `when` clauses.

Template 1: Restrict PII Access

```json
{
  "name": "restrict-pii-access",
  "description": "Prevent agents from accessing customer PII without explicit approval",
  "rules": [
    {
      "action": "read",
      "resource_pattern": "customers/*/pii",
      "effect": "deny",
      "conditions": {
        "unless": {"capability": "pii-approved"}
      }
    }
  ]
}
```

Use case: data-protection programs (for example GDPR work and SOC 2 CC6.1). Only agents that carry the `pii-approved` capability can read personal data; everyone else is denied.

Template 2: Restrict Financial Operations

```json
{
  "name": "restrict-financial-ops",
  "description": "Limit financial operations to authorized agents with high trust",
  "rules": [
    {
      "action": "transfer",
      "resource_pattern": "accounts/*",
      "effect": "deny",
      "conditions": {
        "unless": {"capability": "financial-ops"},
        "min_trust_score": 0.9
      }
    },
    {
      "action": "delete",
      "resource_pattern": "accounts/*",
      "effect": "deny"
```

```
    }
  ]
}
```

Use case: financial services controls. Transfers require both the `financial-ops` capability AND a trust score above 0.9. Account deletion is always denied, with no exemption.

Template 3: Business Hours Only

```json
{
  "name": "block-after-hours-data-export",
  "description": "Block data exports outside business hours",
  "rules": [
    {
      "action": "export",
      "resource_pattern": "reports/*",
      "effect": "deny",
      "conditions": {
        "time_window": {"after": "17:00", "before": "09:00", "timezone":
"America/New_York"}
      }
    }
  ]
}
```

Use case: reduce the window for after-hours data exfiltration. The deny rule is active outside 9am to 5pm Eastern Time.

Template 4: Rate-Limited API Access

```json
{
  "name": "api-rate-guard",
  "description": "Prevent agents from hammering external APIs",
  "rules": [
    {
      "action": "api_call",
      "resource_pattern": "external/*",
      "effect": "deny",
      "conditions": {
        "max_rate": {"count": 60, "period": "1m"}
      }
    }
```

```
  ]
}
```

Use case: cost control and good citizenship. No agent can make more than 60 external API calls per minute.

Template 5: Cross-Org Trust Gate

```json
{
  "name": "cross-org-trust-gate",
  "description": "Only allow cross-org interactions with high-trust agents",
  "rules": [
    {
      "action": "*",
      "resource_pattern": "external-org/*",
      "effect": "deny",
      "conditions": {
        "min_trust_score": 0.8
      }
    }
  ]
}
```

Use case: trust federation. Any action against external-org resources requires a trust score of at least 0.8. Cross-organization trust attestations are a hosted-plane capability, so this template is most useful with Qortara Cloud Governance once it is available.

Creating a Policy (hosted, pre-launch)

On the hosted plane, you create a policy by posting the JSON pack to the policy API. The endpoint below is part of the pre-launch design.

```bash
curl -X POST https://api.qortara.com/v1/policy/policies \
  -H "$AUTH_HEADER" \
  -H "Content-Type: application/json" \
  -d @policy.json
```

With the open-source sidecar there is no API call: you commit the YAML profile to your repository and reference it by name when you build the agent.

Policy Versioning (hosted, pre-launch)

On the hosted plane, every policy update creates a new version. The previous version is deactivated but preserved in history, so a change can always be traced back.

```bash
# Update an existing policy (creates version 2)
curl -X PUT https://api.qortara.com/v1/policy/policies/pol_789ghi \
  -H "$AUTH_HEADER" \
  -H "Content-Type: application/json" \
  -d @updated-policy.json
```

```json
{
  "policy_id": "pol_789ghi",
  "name": "restrict-pii-access",
  "version": 2,
  "previous_version": 1,
  "status": "active"
}
```

To view the full history: `GET /v1/policy/policies/pol_789ghi/versions`.

With the sidecar, your version history is your version control system. Keep profiles in Git so every change has an author, a timestamp, and a reviewable diff.

Testing a Policy Before Deploying

Test a policy against a hypothetical scenario before it affects a running agent.

On the hosted plane (pre-launch), the evaluation endpoint runs a context against your active policies and returns the decision without changing agent behavior:

```bash
curl -X POST https://api.qortara.com/v1/policy/evaluate \
  -H "$AUTH_HEADER" \
  -H "Content-Type: application/json" \
  -d '{
    "context": {
      "agent_did": "did:qor:agent:test-agent",
      "action": "read",
      "resource": "customers/12345/pii",
      "metadata": {"dry_run": true}
    }
  }'
```

The `dry_run` flag is for your own tracking. Qortara evaluates the same way regardless, and it still records a governance event, which is also a convenient way to test your SIEM integration.

With the open-source sidecar, you test by running the agent against an allow case and a deny case in your own test suite, and asserting on the decision:

```python
from qortara_governance_langchain import PolicyDenied

def test_pii_read_is_denied():
    try:
        agent.invoke({"input": "Read PII for customer 12345"})
        assert False, "expected the read to be denied"
    except PolicyDenied as denied:
        assert denied.decision.effect == "deny"
```

Best Practices

1. **Start with deny-by-default.** Create a catch-all deny, then add specific allow rules for approved actions. This is the zero-trust posture: an action is governed unless you have explicitly permitted it.

2. **Use capabilities, not agent names.** Policies that reference capabilities (`pii-approved`, `admin`, `financial-ops`) survive personnel and fleet changes. Policies that hard-code specific agent identifiers rot quickly.

3. **One policy per concern.** Do not bundle unrelated rules. PII access control and rate limiting belong in separate policies so each one is easy to read, review, and change.

4. **Test before deploying.** Use the dry-run pattern (hosted) or a test case (sidecar) to confirm a policy does what you expect before it touches a running agent.

5. **Monitor first, enforce later.** Where you can run a rule in a warn posture (log the decision but allow the action), do that first. It shows you what would be blocked without actually blocking it, which de-risks the switch to deny.

6. **Fail closed.** If the decision point is unreachable, the safe default is to block the governed action rather than let it through ungoverned. Author and operate your policies so a denied result can never quietly become a permissive one.

Related

- [Policy Enforcement](/docs/concepts/policy-enforcement): how runtime decisions are

made and why pre-execution interception matters.
- [Getting Started](/docs/quickstart): install the sidecar and run a governed agent.
- [Compliance Evidence](/docs/concepts/compliance-evidence): how these decisions become audit-ready records.

Product: Qortara Cloud Governance
Source owner: Qortara
Last reviewed: 2026-06-10
Verified against: Qortara pre-launch docs