AI Agent Integration

Version 1.3.0-SNAPSHOT, 2026-02-13T14:32:21Z

Table of Contents

This section describes APIs to add to the framework to support AI agent integration, aligned with patterns discussed in agentic enterprise systems (e.g. Palantir AIP: session management, retrieval context, and tools for action-taking agents).

Reference: "Agentic Operating System for the Enterprise | Palantir’s AIP Lead Jack Dobson at AIPCon 6".

Chapter 1. Goals

- Let agents **discover** what they can do (tools, schema) without hard-coding endpoint URLs.
- Let agents **execute** governed actions (query, save, delete) with existing security (realm, permissions, `@FunctionalAction`).
- **Multi-tenant:** Each tenant (realm) has its own agent configuration and tool instances; tools list and execute are scoped by tenant.
- **Tenant runAs:** Tenant configuration can specify a `runAsUserId` so tool execution runs under that user's security context (PrincipalContext), integrating with the overall security and policy framework.
- Provide **retrieval context** (ontology, type schemas) so the agent can build valid requests and inject context into the LLM.
- Support **session/trace** for audit and multi-turn correlation.

Chapter 2. Query Gateway as the Single Integration Point (CRUDL)

For **CRUDL** (Create, Read, Update, Delete, List) over entity data, agents and MCP should integrate with the **Query Gateway** only—not across the broader REST API space. The gateway is generic and acts as a single integration point for all Morphia-mapped entity types.

Why the gateway is sufficient:

- **One set of endpoints** — `GET /api/query/rootTypes`, `POST /api/query/plan`, `POST /api/query/find`, `POST /api/query/save`, `POST /api/query/delete`, `POST /api/query/deleteMany` cover List (rootTypes + find), Read (find), Create/Update (save), and Delete (delete, deleteMany).
- **Type-agnostic** — The gateway accepts a `rootType` (e.g. `Location`, `Order`) and a query or entity body; there is no need for the agent to know or call domain-specific resources (e.g. `/orders`, `/products`). The same six operations work for every entity type.
- **Single security surface** — One resource (`QueryGatewayResource`) with one `@FunctionalMapping` (area/domain) and method-level `@FunctionalAction`; permission and realm resolution are consistent for all CRUDL.
- **Simpler agents and bridges** — An AI agent or MCP bridge can implement exactly six tool shapes (rootTypes, plan, find, save, delete, deleteMany) and one base path (`/api/query`). No need to discover or maintain many REST paths per domain.

Implications for agent and MCP design:

- **Tools** — Agent/MCP tools should map directly to gateway operations (query_rootTypes, query_plan, query_find, query_save, query_delete, query_deleteMany). No separate tools per entity or per domain resource.
- **Schema** — Discovery and schema can be gateway-derived: `rootTypes` lists available types; optional per-type schema (e.g. `GET /api/agent/schema/{rootType}`) can be built from the same Morphia metadata the gateway uses.
- **Optional agent layer** — A thin layer (`GET /api/agent/tools`, `POST /api/agent/execute`, `GET /api/agent/schema`) can sit on top of the gateway to provide tool discovery, unified execute, and schema for LLMs; implementation delegates to the gateway and does not duplicate logic.
- **Domain REST resources** — Domain-specific resources (e.g. `BaseResource`/`OntologyAwareResource` subclasses with paths like `/orders`, `/shipments`) remain for human-driven UIs and workflows. Agents and MCP use the gateway for CRUDL unless a future design explicitly adds agent-only tools that call those resources.

See REST CRUD and Planner and Query Gateway for the gateway API and query syntax.

Chapter 3. Realistic Scenarios and Use Cases

The agent APIs and MCP bridge are abstract until you see how they are used in practice. Below are concrete scenarios that show **who** uses the system, **what** they ask, and **how** the tools and schema flow fits in.

3.1. Scenario 1: Support agent looking up a customer or order

Who: A support rep in a chat UI backed by an AI (e.g. ChatGPT, Claude, or an in-app bot). **What they ask:** "What's the status of order #ORD-8842 for customer acme@example.com?"

Flow:

1. The AI client (or your app) connects to an MCP server that is configured to call your Quantum backend (`QUANTUM_BASE_URL`, `QUANTUM_AUTH_TOKEN`).
2. The MCP server exposes six tools. The LLM chooses `query_find` and calls it with something like: `rootType: "Order"`, `query: "orderNumber:ORD-8842"`, `realm: "acme-corp"`.
3. The bridge sends `POST /api/agent/execute` with `tool: "query_find"` and those `arguments`. The backend runs the query in the correct realm with your existing security and data scoping.
4. The LLM gets back a list of matching orders (or none) and answers: "Order ORD-8842 is Shipped, delivered last Tuesday."

Why it matters: Support doesn't need to learn your UI or query syntax. They ask in natural language; the agent uses the **same** gateway and permissions as your human-driven app.

3.2. Scenario 2: Analyst exploring data via natural language

Who: A business analyst in a BI or "ask your data" product (e.g. Cursor, a Slack bot, or an internal Copilot). **What they ask:** "Show me all active locations in the West region, and how many orders each had last month."

Flow:

1. The analyst's client uses MCP to talk to your bridge. The bridge has already called `GET /api/agent/tools` and `GET /api/agent/schema` so the LLM knows there are types like `Location`, `Order`, and that it can run `query_find` with a BIAPI query string.
2. The LLM might first call `query_find` with `rootType: "Location"`, `query: "region:West && status:ACTIVE"`, `realm: "acme-corp"`, `page: { limit: 50 }`.
3. It then calls `query_find` again for `Order` with a query that filters by date and optionally by location (if the schema shows how orders link to locations). Your gateway returns rows; the LLM summarizes or charts them.

Why it matters: The analyst never writes BIAPI or touches `/api/query` directly. Discovery (tools +

schema) is enough for the LLM to build valid `rootType` and `query` arguments.

3.3. Scenario 3: Developer in Cursor asking about live data while coding

Who: A developer working in Cursor (or another IDE with MCP) on a feature that uses your API.
What they ask: "What does a CodeList entity look like in our API?" or "Find me a sample Location so I can see the shape of the response."

Flow:

1. Cursor starts your MCP bridge (e.g. `quantum-mcp-bridge`) with env pointing at your dev/staging backend.
2. The developer asks in the chat. The LLM uses MCP `resources/list` (your bridge → `GET /api/agent/schema`) to see that `CodeList` and `Location` exist.
3. It uses `resources/read` for `CodeList` (bridge → `GET /api/agent/schema/CodeList`) to get the JSON Schema–like shape (fields, types).
4. To get a sample, it calls `tools/call` for `query_find` with `rootType: "Location"`, `query: "status:ACTIVE"`, `page: { limit: 1 }`. The bridge sends `POST /api/agent/execute`; the developer sees a real example in the chat.

Why it matters: The developer stays in the IDE and gets schema + live samples without opening Swagger or running `curl` by hand. The bridge is the only integration point.

3.4. Scenario 4: Tenant-specific bot running as a service user

Who: A tenant (e.g. "acme-corp") has an automated bot that creates or updates records (e.g. syncing from an external system). The bot should run with a fixed service identity for audit and security.

Flow:

1. You configure the tenant: `quantum.agent.tenant.acme-corp.runAsUserId=bot-sync@acme-corp`, and optionally `enabledTools=query_find,query_save`, `maxFindLimit=100`.
2. When the MCP bridge (or any client) calls `POST /api/agent/execute` with `arguments.realm: "acme-corp"`, the backend loads tenant config and sees `runAsUserId`. If your app provides a `RunAsPrincipalResolver`, the backend runs the gateway call **as** `bot-sync@acme-corp` (that user's permissions and data scope).
3. The caller (e.g. an API key for the sync job) only needs permission to call the agent API for that tenant; the actual data access is under the service user. Audit logs can record both caller and runAs user.

Why it matters: Bots and integrations get a clear, tenant-scoped identity and limits without you building separate "bot-only" APIs.

3.5. Scenario 5: One integration test as the “contract” for the bridge

Who: You or a bridge implementer. **What you need:** A single, runnable example that proves the backend supports the full flow a bridge must perform.

Flow:

1. Run the framework’s integration test: `mvn -pl quantum-framework -Dtest=AgentMcpBridgeFlowIT test`.
2. The test does, in order: discover tools (`GET /api/agent/tools`), list schema (`GET /api/agent/schema`), read schema for one type (`GET /api/agent/schema/CodeList`), then execute `query_rootTypes`, `query_plan`, and `query_find` via `POST /api/agent/execute`. That sequence is exactly what an MCP bridge does.
3. If the test passes, the backend contract is satisfied. A TypeScript or Python MCP server that calls the same endpoints with the same shapes will work against this backend.

Why it matters: The test is both a regression guard and a living specification for “how to use the agent API like an MCP bridge.”

3.6. Scenario 6: Suggesting regulatory obligations for an entity in a jurisdiction

Who: A compliance or operations user in the PSA app (e.g. PSM frontend). **What they need:** Given an entity (e.g. Legal Entity) in a jurisdiction/state, see **suggested** regulatory obligations to track; accept the ones that apply and have them created in the database.

Flow:

1. User is on a Jurisdiction or Legal Entity detail page and clicks "Suggest obligations."
2. The frontend calls a **domain** endpoint (e.g. `POST /psa/obligations/suggest`) with jurisdiction (and optional entity). The backend uses jurisdiction requirements (and optionally an LLM) to produce a list of suggested obligations (details, due date, priority).
3. The UI shows the suggestions; the user selects which to accept and clicks "Accept selected."
4. The frontend calls the Query Gateway `POST /api/query/save` (rootType Obligation) for each accepted suggestion, persisting new Obligation records.

Why it matters: The framework’s agent tools (query_find, schema) can feed context into an LLM or rule engine; the "suggest" capability is implemented as a domain REST endpoint in the app (psa-app), not as a seventh gateway tool. User acceptance keeps humans in the loop before data is written. The full design is PSA-specific and lives in the psa-app repository: `doc/design/regulatory-obligations-agent-design.adoc` (Regulatory Obligations — Agent-Driven Suggestions and User Acceptance).

Chapter 4. Integrating the query grammar into the agent framework

Developers and agents need to know **what query string** to use for questions like "retrieve locations in Atlanta" or "get orders with their customer hydrated." The framework exposes the query grammar and **"did you know"** hints so the LLM can both **answer** "what is the query string to retrieve X?" and **suggest** expand/ontology usage.

4.1. Query-hints endpoint

Endpoint: `GET /api/agent/query-hints`

Purpose: Return a structured summary of the BIAPI query grammar, example queries by intent, and "did you know" hints. The MCP bridge or agent can call this once (or on demand) and inject the response into context so the LLM can:

- Answer: "What is the query string to retrieve locations in Atlanta?" → use `exampleQueries` (e.g. `city:Atlanta` or `address.city:Atlanta` for `rootType: Location`).
- Suggest: "Did you know you could use the query gateway with expand on entity Order with relationship customer to get a combined result set?" → use `didYouKnow` (e.g. expand(customer) && status:ACTIVE).
- Use ontology: When the app uses ontology edges (hasEdge, hasEdgeAny, notHasEdge), `queryGrammarSummary.ontologyEdges` and a did-you-know hint explain that ontology-aware list endpoints filter by relationship; for gateway find, use attribute filters.

Response shape:

- `queryGrammarSummary` — Short doc: `syntax`, `operators` (equals, inList, wildcards, andOrNot, etc.), `expand` (expand(path), AGGREGATION mode), `ontologyEdges` (hasEdge usage in list/ontology endpoints).
- `exampleQueries` — List of `{ intent, query, rootType, description }`. Examples: "Locations in Atlanta" → `city:Atlanta`, "Orders with customer hydrated" → `expand(customer) && status:ACTIVE`, "Locations whose name contains 'Warehouse'" → `name:*Warehouse*`.
- `didYouKnow` — List of `{ title, body, exampleQuery, rootTypeExample }`. Hints such as: "You can use expand(path) to get a combined result set", "You can expand array references with [*]", "Check the plan before running a query with expand", "Ontology relationships can filter list results", "Wildcards and numeric/date prefixes".

4.2. Example: "What is the query string to retrieve locations in Atlanta?"

1. The user (or developer in Cursor) asks: "What is the query string to retrieve locations in Atlanta?"
2. The agent has already loaded `GET /api/agent/query-hints` (or loads it now). From `exampleQueries`

it finds an entry with intent "Locations in Atlanta": `query: "city:Atlanta"`, `rootType: "Location"`.

3. The agent answers: "Use `city:Atlanta` with `rootType: Location`. If your Location model uses a different field (e.g. address.city), use `address.city:Atlanta`. Call query_find with arguments: rootType: Location, query: city:Atlanta, page: { limit: 10 }."
4. Optionally the agent can run query_plan or query_find to validate or return a sample.

4.3. Example: "Did you know you could use expand on Order with customer?"

1. The agent surfaces a hint from `didYouKnow`: "You can use expand(path) to get a combined result set. Instead of fetching an order and then its customer in a second call, use expand(customer) in the query string. Example: expand(customer) && status:ACTIVE for rootType Order."
2. The developer (or another agent) can then use `query_find` with `query: "expand(customer) && status:ACTIVE"` and `rootType: "Order"` to get orders with nested customer objects. If AGGREGATION execution is enabled, the result set looks like: each order has a `customer` field populated with the related entity.

4.4. Expand and ontology in the same flow

- **expand(path)** — Use in the **query string** passed to `query_find` or `query_plan`. The gateway parses expand(...) and selects AGGREGATION mode; execution (when enabled) returns a combined result set. See Relationship hydration with expand(path) and Planner and QueryGateway.
- **Ontology edges (hasEdge, hasEdgeAny, notHasEdge)** — Used by ontology-aware list endpoints and permission rules (ListQueryRewriter). For the **generic** query gateway find, use normal attribute filters. For entities that have ontology list endpoints (e.g. GET /locations/ontology), those endpoints accept ontology constraints; the agent can point the user to the right endpoint or use query_find with attribute filters only.

In short: expose `GET /api/agent/query-hints` so developers and agents can ask "what query gets X?" and get "did you know you could use expand(...) to get a combined result set that looks like this...".

Chapter 5. Asking permission and policy questions

Agents and developers need to answer: **"Can role X perform this action on this Entity?"**, **"If not, why not—which rule caused the denial?"**, **"What rule changes would allow this identity/action/entity combination?"**, and **"What least-privilege rules are needed for this scenario?"**. The framework exposes a **permission check API** and an **evaluate API** so the LLM can answer these questions and suggest minimal policy changes.

5.1. Permission check API (single decision)

Endpoint: `POST /system/permissions/check` (see Permissions).

Purpose: Determine whether a given identity (userId or role) is allowed to perform a given action in a given area/functionalDomain. The response includes the **winning rule** that determined the outcome, so when the decision is DENY you can answer "why not? — rule R caused it."

Request body (CheckRequest):

- `identity` (required): userId or role name (e.g. `support@acme.com` or `ADMIN`). Roles are resolved from credential, user profile groups, and token.
- `area`, `functionalDomain`, `action` (required for a concrete check): Map from the REST capability you care about. For the Query Gateway: `area=integration`, `functionalDomain=query`, `action=find|save|delete|listRootTypes|plan|deleteMany`. For other resources, use the `@FunctionalMapping` / `@FunctionalAction` values (e.g. `area=MIGRATION`, `functionalDomain=INDEXES`, `action=APPLY_ALL_INDEXES`).
- Optional: `realm`, `tenantId`, `orgRefName`, `accountNumber`, `dataSegment`, `ownerId`, `resourceId` for DataDomain/body matching.
- Optional: `modelClass`, `resource` (JSON snapshot) for in-memory filter evaluation when the rule has andFilterString/orFilterString.

Response (SecurityCheckResponse):

- `decision`: `"ALLOW"` or `"DENY"`.
- `finalEffect`, `decisionScope`: `EXACT` | `SCOPED` | `DEFAULT`.
- **`winningRuleName`**: The rule that determined the outcome. When `decision` is DENY, this is the rule that caused the denial—answer "if not, why not?" with this name.
- `winningRulePriority`, `winningRuleFinal`: Metadata of the winning rule.
- `scopedConstraints`, `filterConstraints`, `notApplicable`: Optional details for SCOPED or filter-related decisions.

Example: "Can role ABC perform find on Location?"

1. Map "find on Location" to the Query Gateway: area=`integration`, functionalDomain=`query`, action=`find`.

2. Call `POST /system/permissions/check` with body: `{"identity": "ABC", "area": "integration", "functionalDomain": "query", "action": "find", "realm": "defaultRealm"}`.
3. If response `decision` is ALLOW → "Yes, role ABC can perform find." If DENY → "No. The rule that caused the denial is: " + response `winningRuleName` (e.g. `default-deny` or a specific DENY rule).

5.2. Evaluate API (full allow/deny matrix for an identity)

Endpoint: `POST /system/permissions/evaluate`

Purpose: Get allow/deny decisions for **all** area/functionalDomain/action combinations (or a subset) for a given identity. Use this to compare "what APIs does this scenario need?" with "what does this identity have?" and identify **gaps** for least-privilege.

Request (EvaluateRequest): `identity` (required), optional `realm`, `roles`, DataDomain fields, and optional filters `area`, `functionalDomain`, `action` to narrow.

Response (EvaluationResult): `allow` and `deny` (area → domain → list of actions), and `decisions[area][domain][action]` with `effect`, `decisionScope`, **`rule`** (winning rule name), `priority`, `finalRule`.

5.3. "If not, why not?" — identify the rule that caused the denial

1. Call `POST /system/permissions/check` with the identity and the desired area/functionalDomain/action.
2. If `decision` is DENY, the response field **`winningRuleName`** is the name of the rule that caused the denial (e.g. `default-deny`, or a DENY rule that matched by priority). Use it to answer "Rule X caused the denial; to allow this combination you would need to add an ALLOW rule with higher priority or narrow the scope of the DENY rule."

5.4. Suggesting changes to rules to allow an action/entity pair

1. **Check** — Run the check for the identity and the desired (area, functionalDomain, action). If DENY, note `winningRuleName`.
2. **Suggest** — Propose a minimal ALLOW rule: identity (or role), area, functionalDomain, action, optional DataDomain/body, priority (e.g. 500 for standard ALLOWs), and optional andFilterString for data scoping. Example: "Add a rule: name=allow-support-query-find, identity=SUPPORT, area=integration, functionalDomain=query, action=find, effect=ALLOW, priority=500."
3. **Authoring** — Actual rule creation/update is done in your rule store (YAML, repo, or admin API). The agent only suggests the rule shape; a human or governance workflow typically applies the change.

5.5. Least-privilege rules for a given scenario

Goal: Determine the set of REST APIs (area/functionalDomain/action) needed to complete a scenario, compare to what a given identity is allowed, and suggest the **minimal** ALLOW rules to close gaps.

Workflow:

1. **Scenario → required APIs** — From the scenario description, list the operations (e.g. "support user must list locations, then find orders for a customer"). Map each to (area, functionalDomain, action): e.g. list locations → integration/query/listRootTypes and integration/query/find with rootType=Location; find orders → integration/query/find with rootType=Order.
2. **Current policies** — Call `POST /system/permissions/evaluate` for the identity that would execute (e.g. `support@acme.com` or role `SUPPORT`). You get allow/deny per (area, domain, action), and per-action `rule` (winning rule name).
3. **Gaps** — Required (area, domain, action) pairs that are DENY or not present in the allow matrix are gaps. Optionally run `POST /system/permissions/check` per gap to get `winningRuleName` for each denial.
4. **Suggest minimal rules** — For each gap, suggest one ALLOW rule (identity, area, functionalDomain, action, priority). Prefer a single role-based ALLOW with the minimum scope (e.g. only the actions needed) rather than broad wildcards. Example: "Add ALLOW rule for identity=SUPPORT, area=integration, functionalDomain=query, action in [listRootTypes, find], priority=500."

Example: Scenario = "Support user lists locations and finds orders by customer ref." Required: (integration, query, listRootTypes), (integration, query, find). Evaluate for identity=SUPPORT → if find is DENY with winningRule=default-deny, suggest: "Add rule allow-support-query-read: identity=SUPPORT, area=integration, functionalDomain=query, action=* (or listRootTypes,find), effect=ALLOW, priority=500."

5.6. Permission hints endpoint (discoverable by agents)

Endpoint: `GET /api/agent/permission-hints`

Purpose: Return a short description of the check and evaluate APIs, how to map "entity" and "action" to area/functionalDomain/action (e.g. Query Gateway mapping), example check requests, and "did you know" hints so the agent can answer permission questions and suggest rule changes. See the Summary Table and MCP Bridge Configuration for the optional call from the bridge.

Chapter 6. Multi-Tenancy and Tenant-Specific Configuration

Agent integration is **multi-tenant**: each tenant (realm) has its own configuration and tool instances.

- **Realm scope** — Tools list and execute are scoped by tenant (realm). The effective realm is resolved from the request (e.g. `realm` query param, `arguments.realm`, or `X-Realm` header), then the principal's default realm, then configured default.
- **Per-tenant configuration** — Each tenant can have:
- *runAsUserId* (optional): The userId to use as the security context when executing agent tools for that tenant. When set, the backend runs the gateway under that user's PrincipalContext so that permissions and data scoping apply as that user (same pattern as impersonation). This integrates with the overall security and policy framework.
- *enabledTools* (optional): List of tool names enabled for that tenant; GET /api/agent/tools returns only those tools for the requested realm.
- *limits* (optional): Per-tenant limits (e.g. max find limit).
- **Caller permission** — The caller must be allowed to use the agent API for the target tenant; the actual execution runs under the tenant's runAs user (if configured) or the caller. Audit logs record both caller and runAs user when runAs is used.

Secrets (global vs tenant-level): Application-wide secrets (e.g. JWT signing key, vault connection) are **global** and tenant-independent; they are resolved at startup. Per-tenant secrets (e.g. LLM API keys for hosted providers like OpenAI, Claude, or Gemini) are **tenant-level** and must be resolved per realm at request time—prefer a vault with realm-scoped paths or a tenant config store that references vault keys. See Secrets and Vault Configuration for the two-level model and how to configure HashiCorp Vault or AWS Secrets Manager.

Token consumption and allocation: Per-tenant token pools can be assigned to different sets of APIs and Tools/LLM configurations for usage-based billing. Record LLM usage (by tenant and runAsUserId) and API call usage (by tenant and caller); allocate token types (API_CALL, LLM_REQUEST, LLM_INPUT_TOKENS, LLM_OUTPUT_TOKENS) scoped to API identifiers and/or tool names and LLM config keys. See Usage Metering and Token Allocation.

See the implementation design (`docs/design/AI_AGENT_INTEGRATION_DESIGN.md`) for the data model (e.g. AgentTenantConfig), storage options, and runAs resolution.

Chapter 7. APIs to Add (Gateway-Centric)

7.1. 1. Agent Tools / Capabilities (Discovery)

Purpose: One endpoint that returns a machine-readable list of "tools" the agent can call—scoped to **Query Gateway operations only** so the agent has a single, consistent CRUDL surface.

Suggested endpoint: `GET /api/agent/tools` (or `GET /api/agent/capabilities`).

Response shape (conceptual):

- List of tools, one per gateway operation:
- `query_rootTypes` — list entity types (maps to `GET /api/query/rootTypes`).
- `query_plan` — get execution plan for a query (maps to `POST /api/query/plan`).
- `query_find` — list/read entities (maps to `POST /api/query/find`).
- `query_save` — create or update an entity (maps to `POST /api/query/save`).
- `query_delete` — delete one entity by id (maps to `POST /api/query/delete`).
- `query_deleteMany` — delete many by query (maps to `POST /api/query/deleteMany`).
- Each tool includes: `name`, `description` (for the LLM), `parameters` (JSON Schema for the request body), and optional `area`/`domain`/`action` for permission alignment.

Implementation notes:

- Implement by enumerating the six gateway operations (and their request DTOs) from `QueryGatewayResource`; no need to scan the rest of the REST API space.
- Only include tools the current principal is allowed to use (e.g. permission for integration/query and the corresponding action).
- Response format can align with OpenAI function calling or MCP tool definitions so agents and MCP bridges consume it directly.

Why: Agents discover a single CRUDL surface (the gateway) instead of many domain endpoints; aligns with "Tools" in Palantir AIP while keeping integration simple.

7.2. 2. Schema / Ontology for Agents (Retrieval Context)

Purpose: Expose type schemas (and optional ontology metadata) so the agent can build valid gateway payloads: `rootType`, `query`, and entity bodies. All schema is **gateway-derived**—the same types the gateway already uses.

Suggested endpoints:

- `GET /api/agent/schema` — list of entity types; **reuse or wrap `GET /api/query/rootTypes`** so there is a single source of truth. Optionally add per-type field metadata (name, type, required).

- `GET /api/agent/schema/{rootType}` — JSON Schema (or equivalent) for one root type (fields, types, constraints) so the agent can construct valid `find`/`save` bodies. Derive from the same Morphia `EntityModel` the gateway uses.
- Optional: `GET /api/agent/ontology` — high-level ontology summary (entity kinds, relations) for retrieval context; can wrap ontology modules if present, but CRUDL schema stays gateway-centric.

Implementation notes:

- Schema should be derived from the same Morphia mapping the gateway uses (e.g. `EntityModel`, `@Property`, `@Reference`); no separate schema store.
- Keep responses concise so they fit in LLM context windows.
- Agents that only use the gateway need only `rootTypes` and optional per-type schema; ontology is an optional enrichment.

Why: Single source of truth (gateway + Morphia); agents get valid `rootType` and field names without integrating with domain-specific REST docs.

7.3. 3. Unified Agent Action API (Optional)

Purpose: Single HTTP entry point for the agent to execute a gateway "tool" by name and arguments, so the agent does not need to know the six gateway paths. The implementation **delegates only to the Query Gateway**.

Suggested endpoint: `POST /api/agent/execute` (or `POST /api/agent/tools/execute`).

Request shape (conceptual):

- `tool` (or `name`): one of `query_rootTypes`, `query_plan`, `query_find`, `query_save`, `query_delete`, `query_deleteMany`.
- `arguments` (or `params`): JSON object matching the gateway request body for that operation (e.g. `rootType`, `query`, `page` for `query_find`; `rootType`, `entity` for `query_save`).
- Optional: `sessionId`, `traceId` for audit and multi-turn correlation.

Response: Same as the underlying gateway operation (e.g. find result, save result, delete result, or error).

Implementation notes:

- **Delegate only to `QueryGatewayResource`** — no branching to domain resources. Map `tool` to the corresponding gateway method (plan, find, save, delete, deleteMany, rootTypes) and pass `arguments` as the request body. One integration point, one security surface.
- Security: same realm resolution, permission checks (`@FunctionalAction`), and data scoping as the gateway.

- Optional: validate `tool` and `arguments` against the schema from `GET /api/agent/tools` to return clear errors.

Why: Agent calls one execute endpoint; the server translates to the appropriate gateway call. No need for the agent (or MCP bridge) to integrate across the REST API space for CRUDL.

7.4. 4. Session / Trace Headers (Audit and Multi-Turn)

Purpose: Allow agent runtimes to pass a session and/or trace ID so that all requests belonging to one conversation or run can be correlated and audited.

Suggested approach:

- Request headers: e.g. `X-Agent-Session-Id`, `X-Agent-Trace-Id` (or `X-Request-Trace-Id`).
- Store or log these in existing audit / request logging so that support and compliance can filter by session or trace.
- No new endpoint required; only header handling and propagation to logging or audit tables.

Implementation notes:

- Security filter or request filter: read headers, put values into `SecurityContext` or request-scoped context so that any downstream code (e.g. query gateway, permission checks) can attach them to logs or audit events.
- Optional: include `sessionId` / `traceId` in error responses (e.g. in a response header or a small JSON envelope) so the agent runtime can correlate failures.

Why: Matches multi-turn and session management in agentic APIs; supports debugging and compliance.

7.5. 5. Retrieval Context Endpoint (Optional, RAG-Like)

Purpose: Allow the agent to request a chunk of "context" (e.g. entities or ontology neighborhood) to inject into the LLM. Prefer **gateway-based** context where possible.

Suggested endpoint: `POST /api/agent/context` (or `GET /api/agent/context` with query params).

Request (conceptual):

- `type`: e.g. `entity_by_id`, `search`, or optional `ontology_edges`.
- Parameters: e.g. `rootType`, `query`, `limit`, `id` (for entity_by_id); optional `entityId` for ontology edges.

Response: Structured context (e.g. list of entities, or ontology edges) in a stable, concise format.

Implementation notes:

- **Gateway-first** — For `entity_by_id` or `search`, implement by calling the gateway: e.g. `POST /api/query/find` with `rootType`, `query` (e.g. `_id:…` for by-id), and a small `limit`. No need to call domain-specific list endpoints.
- Optional: for `ontology_edges`, wrap `OntologyAwareResource`/ontology services if the app uses them; otherwise omit or keep context to gateway find results.
- Limit size (e.g. `limit` cap) to avoid overwhelming the LLM context window.
- Enforce same security and realm as the gateway.

Why: Retrieval context stays aligned with the single integration point (gateway find); optional ontology enriches when available.

Chapter 8. MCP Integration (Cursor, ChatGPT, Gemini)

The Model Context Protocol (MCP) is an open standard that lets AI clients (Cursor, Claude Desktop, ChatGPT, Gemini, etc.) discover and call **tools** and read **resources** from a server. Integrating MCP with the **Query Gateway as the single integration point** means the MCP bridge talks only to the gateway (and optional thin agent layer) for CRUDL—no integration across the rest of the REST API space.

8.1. How MCP Maps to the Framework (Gateway-Centric)

MCP concept	Purpose	Framework equivalent (gateway-only for CRUDL)
Tools	Actions the LLM can call (`tools/list`, `tools/call`)	Six tools: `query_rootTypes`, `query_plan`, `query_find`, `query_save`, `query_delete`, `query_deleteMany` → `GET /api/query/rootTypes`, `POST /api/query/plan`, `POST /api/query/find`, etc. (or `POST /api/agent/execute`)
Resources	Read-only data for context (`resources/list`, `resources/read`)	`GET /api/query/rootTypes` or `GET /api/agent/schema`; `GET /api/agent/schema/{rootType}` for per-type schema. No need to expose domain REST paths as resources.
Prompts (optional)	Parameterized prompt templates	Domain prompts that invoke the six gateway tools (e.g. "List active locations", "Find orders for customer X")

MCP uses JSON-RPC over stdio (local) or HTTP (SSE / Streamable HTTP) for remote servers. Cursor and Claude support both; ChatGPT and Gemini can use remote MCP when they support the protocol.

8.2. Two Integration Approaches

8.2.1. Option A: MCP Bridge Server (recommended to start)

Run a **standalone MCP server** that talks to your Quantum REST APIs. The AI client (Cursor, Claude, etc.) connects to the MCP server; the MCP server calls your backend (e.g. `https://your-app.example.com`).

Pros:

- No changes to the Java/Quarkus app; reuse existing REST APIs (and future `/api/agent/*` endpoints).
- Same security: backend still enforces realm, permissions, and `@FunctionalAction`.
- Works with Cursor (stdio or remote SSE), Claude Desktop (stdio), Claude in browser (Custom Connector / remote), and any MCP client.
- Can be implemented in TypeScript, Python, or another language using official MCP SDKs.

How it works:

1. **MCP tools** — The bridge implements exactly **six tools** (query_rootTypes, query_plan, query_find, query_save, query_delete, query_deleteMany), each mapping to one Query Gateway endpoint. No tools for domain resources (e.g. no `/orders` or `/products`). `tools/list` can call `GET /api/agent/tools` when available, or return a fixed list of these six. `tools/call` forwards to `POST /api/agent/execute` or directly to `POST /api/query/find`, `POST /api/query/save`, etc., with the same JSON bodies. **Single integration point:** one base path (`/api/query`) and optionally `/api/agent/*`.
2. **Auth** — The bridge is configured with a base URL and an auth token (env). Every request to the backend includes that token. The backend (gateway) authenticates and authorizes as for any REST client.
3. **MCP resources** (optional) — Expose gateway-derived schema only: e.g. `quantum://schema` → `GET /api/query/rootTypes` (or `GET /api/agent/schema`), `quantum://schema/{rootType}` → `GET /api/agent/schema/{rootType}`. No need to expose domain REST paths as MCP resources for CRUDL.

Example tool mapping (bridge → backend):

MCP tool name	Bridge action
`query_rootTypes`	`GET /api/query/rootTypes`
`query_plan`	`POST /api/query/plan` with `rootType`, `query`
`query_find`	`POST /api/query/find` with `rootType`, `query`, `page`, `sort`, optional `realm`
`query_save`	`POST /api/query/save` with `rootType`, `entity`, optional `realm`
`query_delete`	`POST /api/query/delete` with `rootType`, `id`, optional `realm`
`query_deleteMany`	`POST /api/query/deleteMany` with `rootType`, `query`, optional `realm`

Where to put the bridge:

- **Separate repo or module** (e.g. `quantum-mcp-bridge`): TypeScript or Python app that uses the MCP SDK; **only** calls the Query Gateway (and optional agent schema/execute) — one base URL, six operations. Configurable base URL and auth via env; run locally (stdio) or as HTTP MCP (SSE) for remote clients.
- **Same repo, optional package**: e.g. `quantum-mcp-bridge/` with its own `package.json` or `pyproject.toml`; document that CRUDL integration is gateway-only so adopters do not add per-domain REST calls.

8.2.2. Option B: Native MCP in the Backend

Expose an MCP-compatible HTTP endpoint (e.g. Streamable HTTP or SSE) from the Quarkus application. The MCP endpoint would speak the MCP JSON-RPC protocol and delegate **all tool calls to the Query Gateway** (and resource reads to gateway-derived schema). No delegation to domain REST resources for CRUDL.

Pros: Single deployment; no separate bridge process. **Cons:** Requires an MCP server implementation in Java (or a small sidecar); more invasive. Can be a later phase once the gateway-centric agent APIs are stable.

8.3. Client Configuration

Cursor

- Project-level: create `.cursor/mcp.json` in the repo.
- For a **local** bridge (stdio): point to the bridge command (e.g. `npx` / `node` or `uv run` for Python).
- For a **remote** bridge (SSE): use the SSE transport and the bridge's URL (e.g. `https://mcp-bridge.example.com/sse`).

Example (local bridge, stdio):

```
{
  "mcpServers": {
    "quantum": {
      "command": "node",
      "args": ["path/to/quantum-mcp-bridge/build/index.js"],
      "env": {
        "QUANTUM_BASE_URL": "https://your-api.example.com",
        "QUANTUM_AUTH_TOKEN": "<bearer-or-api-key>"
      }
    }
  }
}
```

Claude (Desktop) — Same idea: add the bridge under `mcpServers` in the Claude Desktop config (e.g. `~/Library/Application Support/Claude/claude_desktop_config.json` on macOS), with `command`/`args` for stdio or use Custom Connector for a remote URL.

Claude (browser) — Use Custom Connectors: add the remote MCP server URL (your bridge's SSE endpoint) and complete authentication as required by your bridge (e.g. API key or OAuth).

ChatGPT / Gemini — When they support MCP, use the remote (SSE) URL of your bridge and any documented auth (header or OAuth). No changes needed on the Quantum backend.

8.4. Auth and Security (Bridge)

- **Token storage:** The bridge should receive the backend auth token via environment variable (or a secure config), not hardcoded. Cursor/Claude inject env when starting the bridge (e.g. `QUANTUM_AUTH_TOKEN`).
- **Backend unchanged:** Realm resolution, permission checks, and data scoping remain in the framework; the bridge is a transparent HTTP client.
- **Optional:** Support `X-Realm` (or equivalent) in tool arguments so the user/agent can scope requests to a tenant when allowed by the principal.

8.5. MCP Bridge Configuration

This section defines the **configuration contract** between an MCP bridge and the Quantum backend. A bridge that implements this contract can discover tools, read schema, and execute gateway operations via the agent REST APIs. The framework provides integration tests (e.g. `AgentMcpBridgeFlowIT`) that exercise this flow as a living example.

8.5.1. Environment Variables (Bridge)

The bridge must be configured with at least the backend base URL and an auth token. All HTTP requests from the bridge to the backend use these values.

| Variable | Required | Description | |----------|----------|-------------| | `QUANTUM_BASE_URL` | Yes | Base URL of the Quantum REST API (e.g. `https://your-app.example.com`). No trailing slash. | | `QUANTUM_AUTH_TOKEN` | Yes | Bearer token or API key for authentication. Sent as `Authorization: Bearer <token>` (or as required by the app). | | `QUANTUM_REALM` | No | Default realm for tool calls when the user/agent does not pass `realm` in arguments. When set, the bridge can add `realm` to every execute payload. |

8.5.2. HTTP Endpoints the Bridge Calls

The bridge implements MCP `tools/list`, `tools/call`, and optionally `resources/list` / `resources/read` by calling the following agent endpoints:

- **MCP `tools/list`** → `GET {baseUrl}/api/agent/tools?realm={realm}` Returns `{ "tools": [ { "name", "description", "parameters", "area", "domain", "action" }, … ], "count" }`. Use optional `realm` when `QUANTUM_REALM` is set or the client requests a tenant.
- **MCP `tools/call`** → `POST {baseUrl}/api/agent/execute` Body: `{ "tool": "<name>", "arguments": { "rootType"?, "query"?, "entity"?, "id"?, "realm"?, "page"?, "sort"?, … }, "sessionId"?, "traceId"? }`. Response: same as the underlying gateway operation (e.g. find returns a collection, save returns save response). The bridge forwards the MCP tool name (e.g. `query_find`) and the MCP arguments map as `arguments`.
- **MCP `resources/list`** (optional) → `GET {baseUrl}/api/agent/schema` Returns `{ "rootTypes": [ { "className", "simpleName", "collectionName" }, … ], "count" }`. The bridge can expose these as MCP resources (e.g. `quantum://schema` or one resource per root type).
- **MCP `resources/read`** for a type (optional) → `GET {baseUrl}/api/agent/schema/{rootType}` Returns JSON Schema-like `{ "type", "title", "properties", … }` for the given root type (simple name or FQCN).
- **Query hints** (optional) → `GET {baseUrl}/api/agent/query-hints` — Returns query grammar summary, example queries (e.g. "Locations in Atlanta" → `city:Atlanta`), and "did you know" hints (e.g. expand(customer) for combined result sets). Inject into LLM context so the agent can answer "what is the query string to retrieve X?" and suggest expand/ontology usage.

All requests must include the auth header (e.g. `Authorization: Bearer <QUANTUM_AUTH_TOKEN>`). Optional headers for audit: `X-Agent-Session-Id`, `X-Agent-Trace-Id`.

8.5.3. Request/Response Examples (Bridge → Backend)

Discover tools (MCP tools/list):

```
GET /api/agent/tools?realm=my-tenant
Authorization: Bearer <token>

Response 200:
```

```
{
  "tools": [
    { "name": "query_rootTypes", "description": "...", "parameters": { "type":
"object", "properties": {}, "required": [] }, "area": "integration", "domain":
"query", "action": "listRootTypes" },
    { "name": "query_find", "description": "...", "parameters": { ... }, "area":
"integration", "domain": "query", "action": "find" },
    ...
  ],
  "count": 6
}
```

Execute a tool (MCP tools/call for query_find):

```
POST /api/agent/execute
Authorization: Bearer <token>
Content-Type: application/json

{
  "tool": "query_find",
  "arguments": {
    "rootType": "CodeList",
    "query": "refName:ACTIVE",
    "realm": "my-tenant",
    "page": { "limit": 10, "skip": 0 }
  },
  "sessionId": "sess-123",
  "traceId": "trace-456"
}

Response 200: (same as POST /api/query/find; Collection envelope)
{
  "rows": [ ... ],
  "offset": 0,
  "limit": 10,
  "filter": "refName:ACTIVE",
  "rowCount": 0
}
```

List schema (MCP resources/list):

```
GET /api/agent/schema
Authorization: Bearer <token>

Response 200:
{
  "rootTypes": [ { "className": "com.example.CodeList", "simpleName": "CodeList",
"collectionName": "codeLists" }, ... ],
  "count": 12
}
```

Read schema for one type (MCP resources/read):

```
GET /api/agent/schema/CodeList
Authorization: Bearer <token>

Response 200:
{
  "type": "object",
  "title": "CodeList",
  "properties": { "refName": { "type": "string" }, "name": { "type": "string" }, ... }
}
```

8.5.4. Cursor Configuration Example

Create `.cursor/mcp.json` (or use Cursor MCP settings) to point at your Quantum MCP bridge. The bridge is a separate process (Node/Python) that reads `QUANTUM_BASE_URL` and `QUANTUM_AUTH_TOKEN` from the environment.

Local bridge (stdio):

```
{
  "mcpServers": {
    "quantum": {
      "command": "node",
      "args": ["/path/to/quantum-mcp-bridge/dist/index.js"],
      "env": {
        "QUANTUM_BASE_URL": "https://api.mycompany.com",
        "QUANTUM_AUTH_TOKEN": "your-bearer-token",
        "QUANTUM_REALM": "defaultRealm"
      }
    }
  }
}
```

Remote bridge (SSE): If the bridge is deployed as an HTTP MCP server (e.g. `https://mcp-bridge.mycompany.com/sse`), use the SSE transport and omit `command`/`args`; configure the remote URL and auth as required by Cursor.

8.5.5. Claude Desktop Configuration Example

Edit the Claude Desktop config (e.g. `~/Library/Application Support/Claude/claude_desktop_config.json` on macOS):

```
{
  "mcpServers": {
    "quantum": {
      "command": "node",
      "args": ["/path/to/quantum-mcp-bridge/dist/index.js"],
      "env": {
        "QUANTUM_BASE_URL": "https://api.mycompany.com",
        "QUANTUM_AUTH_TOKEN": "your-bearer-token",
        "QUANTUM_REALM": "defaultRealm"
      }
    }
  }
}
```

8.5.6. Backend Configuration (Optional)

When the backend is used by an MCP bridge with tenant-specific agent config, set per-realm options in `application.properties` (or equivalent):

```
# Per-tenant agent config (optional)
quantum.agent.tenant.defaultRealm.runAsUserId=agent-service-user
quantum.agent.tenant.defaultRealm.enabledTools=query_rootTypes,query_plan,query_find,q
uery_save
quantum.agent.tenant.defaultRealm.maxFindLimit=100
```

The bridge does not need to know these values; the backend applies them when handling `/api/agent/tools` and `/api/agent/execute`.

8.5.7. Integration Test as Example

The framework includes an integration test that **simulates the MCP bridge flow** end-to-end: discover tools, list schema, read schema for a type, then execute `query_rootTypes`, `query_plan`, and `query_find`. This test serves as a **living example** of the exact HTTP sequence a bridge must perform. See `AgentMcpBridgeFlowIT` in the `quantum-framework` module. Run it with:

```
mvn -pl quantum-framework -Dtest=AgentMcpBridgeFlowIT test
```

8.6. Implementation Order for MCP

1. Implement the **agent REST APIs** (tools discovery, schema, optional execute) as in the Summary Table.
2. Build and document a **bridge** (Option A): one MCP server that implements tools (and optionally resources) by calling those REST APIs; document base URL and auth env vars.
3. Add **Cursor/Claude config examples** to the framework docs (e.g. in this guide or a dedicated MCP section).
4. Optionally add **native MCP** in the backend (Option B) later for a single-deployment story.

8.7. MCP References

- Specification: Model Context Protocol
- Server concepts (tools, resources, prompts): Understanding MCP servers
- Build a server: Build an MCP server
- SDKs: MCP SDKs (TypeScript, Python, Java, etc.)
- Remote servers: Connect to remote MCP servers
- Cursor MCP: Cursor – Model Context Protocol

Chapter 9. Summary Table

CRUDL is fully covered by the Query Gateway (`/api/query`). The agent APIs below are a thin discovery/schema/execute layer on top of the gateway; they do not introduce a second CRUDL surface or require integration across domain REST resources.

| API | Method | Purpose | |-----|--------|--------| | `/api/query/*` (existing) | GET/POST | **Single integration point:** rootTypes, plan, find, save, delete, deleteMany for all entity types. | | `/api/agent/tools` | GET | Discover the six gateway tools (names, descriptions, parameters); permission-filtered. | | `/api/agent/schema` | GET | List entity types (wrap or reuse `GET /api/query/rootTypes`) and optional field metadata. | | `/api/agent/schema/{rootType}` | GET | JSON Schema for one root type (gateway-derived Morphia metadata). | | `/api/agent/query-hints` | GET | Query grammar summary, example queries by intent, and "did you know" hints (expand, ontology). Use so agents can answer "what query gets locations in Atlanta?" and suggest expand/ontology usage. | | `/api/agent/permission-hints` | GET | Permission check/evaluate API summary, area/domain/action mapping (e.g. Query Gateway), example check requests, and "did you know" hints. Use so agents can answer "can role X perform action Y on entity Z?" and "if not, why not?" (winningRuleName), suggest rule changes, and generate least-privilege rules for a scenario. | | `/api/agent/ontology` | GET | Optional: ontology summary for retrieval context. | | `/api/agent/execute` | POST | Optional: execute one of the six gateway tools by name + arguments (delegate only to `QueryGatewayResource`). | | `/api/agent/context` | POST/GET | Optional: retrieval context (gateway find or optional ontology). | | (headers) | - | `X-Agent-Session-Id`, `X-Agent-Trace-Id` for audit and multi-turn. |

Chapter 10. Implementation Order

1. **Treat the gateway as the single CRUDL integration point** — agents and MCP use only `/api/query` (and optional agent layer) for CRUDL; no per-domain REST integration.
2. **Tools discovery** (`GET /api/agent/tools`) — enumerate the six gateway operations; enables agents to know what they can call.
3. **Schema for agents** (`GET /api/agent/schema`, `GET /api/agent/schema/{rootType}`) — gateway-derived from `rootTypes` and Morphia metadata.
4. **Session/trace headers** — low effort, high value for audit and debugging.
5. **Unified execute** (`POST /api/agent/execute`) — optional; delegate only to `QueryGatewayResource`.
6. **Retrieval context** (`POST /api/agent/context`) and **ontology summary** — optional; prefer gateway find for context.

Chapter 11. Security and Governance

- All new agent endpoints delegate to the **Query Gateway** for CRUDL; they use the same security filter, realm resolution, and data scoping as `QueryGatewayResource`.
- Tools discovery lists only the six gateway tools and only if the current principal is allowed (e.g. integration/query area and corresponding action).
- Unified execute must not bypass `@FunctionalAction` or permission checks; it calls the same gateway logic—no delegation to domain resources for CRUDL.
- Prefer existing patterns: `PrincipalContext`, `SecurityContext`, and optional `X-Realm` for tenant isolation.

Chapter 12. References

- **Implementation design:** `docs/design/AI_AGENT_INTEGRATION_DESIGN.md` in the framework repo — API specs, DTOs, backend classes, MCP bridge, implementation phases, and testing strategy.
- Palantir Foundry – Use AIP Agents through Foundry APIs: https://palantir.com/docs/foundry/agent-studio/foundry-apis/
- AIP Agent Studio overview: https://www.palantir.com/docs/foundry/agent-studio/overview
- Video: "Agentic Operating System for the Enterprise | Palantir’s AIP Lead Jack Dobson at AIPCon 6"