

# Agents Guidelines for EasyData Development

This document provides **clear instructions and best practices for any AI agent** tasked with maintaining, extending, or improving the EasyData project. By following these rules, agents will produce consistent, secure, and maintainable contributions aligned with the architecture and documents supplied.

## Reference Documents

Before making changes, read and understand the following core documents in the `docs/` directory. They are your **single sources of truth**:

File	Purpose
<code>master_api_contract.md</code>	Defines every API endpoint, request/response schema, streaming phases, and error codes. Always update this first when designing new endpoints.
<code>data_model_schema_context.md</code>	Describes the system and user databases, tables such as <code>audit_logs</code> and <code>training_data</code> , and how they relate to the RAG process.
<code>adr_arch_dec_record.md</code>	Records architectural decisions (ADR). Create a new ADR whenever you introduce a non-trivial design change.
<code>security_permissions_matrix.md</code>	Specifies SQL firewall rules, RBAC roles and permissions, JWT token expiry, and other security policies.
<code>software_requirements_spec.md</code>	Lists functional/non-functional requirements, constraints, and use cases. Always ensure new features meet these requirements.
<code>project_design_document.md</code>	Provides the overall architecture, layered design, workflow, and major services (e.g. <code>VannaService</code> , <code>OrchestrationService</code> ).
<code>guidelines.md</code>	Offers high-level implementation guidelines, folder structure, configuration principles, and the final canonical structure.

When in doubt, consult these documents. If something is missing or ambiguous, record your reasoning in a new ADR and clarify it in the appropriate document.

## General Rules

1. **Contract First:** Define API request/response models in Pydantic (`models/`) and update `master_api_contract.md` before implementing an endpoint. The frontend relies on this contract to stay stable across backend changes.
2. **Structured Monolith Only:** Do not introduce microservices or random module structures. Follow the canonical project structure described in `guidelines.md` and the `project_design_document.md`.
3. **Single Source of Truth (SSOT):** Configuration values **must be read from** `.env` **via** `core/config.py`. Do not hard-code secrets, connection strings, or feature toggles anywhere else. Provide a `.env.example` for new environment variables.
4. **Lazy Loading:** Use the factory pattern in `providers/factory.py` to instantiate only the necessary providers (DB, LLM, vector store) at runtime. Avoid global imports of database drivers or heavy dependencies.
5. **Keep Services Stateless:** No service should maintain state between requests (other than database or vector store). Avoid singletons with stateful behaviour.
6. **Isolation of Concerns:**
  7. `core/` contains configuration, constants, exceptions, and security utilities. It must not import from `api/` or `services/`.
  8. `providers/` implement interfaces defined in `providers/base.py`. Each provider file should only implement one concrete provider.
  9. `services/` encapsulate business logic. They should not depend on FastAPI or HTTP concepts.
  10. `api/` contains FastAPI routes and dependency injection; it must not contain business logic.
  11. `middleware/` implements cross-cutting concerns (logging, rate limiting, performance, security) and must be toggleable via `.env`.
12. **RBAC and Security:** Enforce permissions using roles and scopes defined in `security_permissions_matrix.md`. Never bypass `require_permission` dependencies in routes. Use the SQL firewall and RLS injection described in the design document for every generated SQL.
13. **Streaming Responses:** When implementing the `/api/v1/ask` endpoint, return data via NDJSON as specified (data → chart → summary). Do not send all data at once.
14. **Circuit Breaker & Error Handling:** Use the `CircuitBreaker` utility to wrap calls to external services (LLM, databases). Centralise exception handling in `core/exceptions.py` and return unified JSON responses (`status`, `message`, `data`, `error_code`, `timestamp`).

15. **Testing and Documentation:** Write unit tests for critical logic. If you introduce a new pattern or complex behaviour, create an ADR in `docs/adr/` explaining your choice. Update the SRS if requirements evolve.

## ✂ Development Workflow for Agents

1. **Plan:** Identify which requirement you are addressing. Consult the SRS, API contract, and design document. If the change affects architecture or introduces a new provider, write an ADR (e.g. `docs/adr/ADR-00XX-new-feature.md`).
2. **Update Contracts:** If adding an endpoint, update `master_api_contract.md` with endpoint details, request/response schemas, and error codes. Define corresponding Pydantic models in `app/models/`.
3. **Implement Provider or Service:** Implement the logic in the appropriate `providers/` or `services/` module. Adhere to interfaces in `providers/base.py` and avoid circular imports.
4. **Wire Up API:** Create or modify routes under `app/api/v1/`. Use FastAPI's `Depends` for authentication, permissions, and DI. Do not put business logic in route handlers.
5. **Security & Performance:** Ensure new queries are validated via the SQL guard (`utils/sql_guard.py`). Respect RLS injection and role checks. Use streaming for long-running queries. Add rate limiting if the endpoint is expensive.
6. **Document & Test:** Document how to use the new feature in the design document if needed. Write tests (under `tests/`) and run them. Update `requirements.txt` if new dependencies are needed, and ensure they comply with project policies.
7. **Review & Commit:** Ensure code meets PEP8 and passes linters. Commit changes with descriptive messages and update any affected documentation. Provide new ADR if applicable.

## 📖 Example Agent Task

*Goal:* Add a new endpoint to export audit logs as CSV for administrators.

**Steps:** 1. Check `security_permissions_matrix.md` to ensure only `admin` or `manager` roles have access. 2. Add an entry in `master_api_contract.md` detailing `GET /api/v1/audit/export`, including query parameters for date range and expected CSV media type. 3. Define a `AuditExportRequest` model in `app/models/request.py`. 4. Implement `AuditService.export_logs()` in `services/` to read from the System DB via SQLAlchemy and generate a CSV stream. 5. Create a route in `api/v1/admin.py` that injects user via `verify_token`, checks `Permission.AUDIT_VIEW`, calls the service, and returns a `StreamingResponse` with appropriate headers. 6. Write a new ADR if you decide to introduce a helper for streaming CSV, explaining why. 7. Update tests and documentation.

## Summary

This file guides any AI agent through the development lifecycle of EasyData. By following these practices and referring to the provided documents, agents will avoid architectural drift, implement features securely, and maintain consistency across the project.

---