# Student Data Agent – Design & Architecture Documentation

## 1 Introduction

This document explains, \*\*in prose only\*\*, how the “Student Data Agent” works. It focuses on \*what\* the code does and \*why\* each design choice was made, without reproducing any source code. It is intended for engineers and product stakeholders who want to understand the architecture, data-flow, and reasoning behind the system.

## 2 High-Level Architecture

The solution turns a WebSocket message into a structured, tool-augmented call to OpenAI and back to the client.

```mermaid

graph LR

UA([User Agent – Browser]) -- text/json --> WS[[Django WebSocket Consumer]]

WS --> |parses & dispatches| AG[Agent Executor]

AG --> |LLM Prompt & Tools| LLM[OpenAI Chat Model]

LLM --> |function-call style| DBTools[Database Tools Layer]

DBTools --> |ORM queries| PG[(PostgreSQL DB)]

DBTools --> LLM

LLM --> AG

AG --> WS

WS --> UA

```

\* \*\*User Agent\*\* – Sends natural-language questions.

\* \*\*WebSocket Consumer\*\* – Acts as the gateway, translating messages into agent invocations and relaying responses.

\* \*\*Agent Executor\*\* – Manages prompt assembly, tool binding, scratch-pad formatting, and conversation memory.

\* \*\*OpenAI Chat Model\*\* – Interprets the prompt, decides when to call a tool, and generates JSON output.

\* \*\*Database Tools Layer\*\* – Thin wrappers around Django ORM queries; exposed to the LLM as callable \*tools\*.

\* \*\*PostgreSQL\*\* – Stores students, courses, grades, etc.

## 3 Prompt Strategy

1. \*\*Role Definition\*\* – The \*system message\* describes the agent’s twofold responsibility: (a) precise student-data retrieval and (b) answering GK/Aptitude questions.

2. \*\*Schema Enforcement\*\* – The prompt mandates a \*\*strict JSON schema\*\* with keys such as `student\_name`, `attendance`, etc., guaranteeing machine-readable replies.

3. \*\*Tool Guidance\*\* – Explicit instructions teach the LLM \*which\* tool answers \*which\* question, ensuring deterministic behaviour.

4. \*\*Examples\*\* – Few-shot examples prime the model to mirror the desired structure and wording.

5. \*\*Edge-case Hints\*\* – Clarifies how to react to missing data or ambiguous names.

### Why this matters

\* Reduces hallucination by tightly scoping the agent’s domain.

\* Off-loads business logic (like field selection) to the LLM while keeping data access server-side.

## 4 Tooling Strategy

| Tool (Logical Name) | Purpose | Typical Trigger Phrases | Notes |

|---------------------|---------|-------------------------|-------|

| \*\*get\_student\_details\*\* | Fetch a \*complete\* student profile | “What is the GPA of…” | Serialises the Student object via DRF serializer. |

| \*\*get\_student\_records\*\* | Fetch a \*raw\* Student ORM instance | Internal helper when other tools need a base record. | Hidden from users. |

| \*\*count\_total\_records\*\* | Return counts for students, courses, etc. | “How many courses…”, dashboard stats | Uses a switch to target the right model. |

| \*\*failed\_students\*\* | List students whose performance status is \*Failed\* | “Show failed students” | Uses `select\_related` for efficiency. |

| \*\*topper\_students\_list\*\* | Top-10 by GPA | “Top students” | Orders by `-gpa` and slices. |

| \*\*get\_student\_session\*\* | Pulls session-specific records (attendance, grades…) | “Show internships for …” | Delegates to DRF serializers per relation. |

### Design Principles

\* \*\*Single Responsibility\*\* – Each tool answers \*exactly one\* type of query.

\* \*\*Asynchronous Safety\*\* – All synchronous ORM calls are wrapped with `sync\_to\_async`, preventing event-loop blocking.

\* \*\*Consistent Interface\*\* – Every tool returns plain Python structures (dict/list) that the LLM easily serialises.

## 5 Execution Flow (Sequence Diagram)

```mermaid

sequenceDiagram

participant U as User

participant C as ChatConsumer

participant A as AgentExecutor

participant L as OpenAI LLM

participant T as Tools Layer

participant DB as Database

U->>C: "What is the attendance of Alexis?"

C->>A: invoke agent

A->>L: system / user / scratchpad messages

L-->>A: function-call get\_student\_details(name="Alexis")

A->>T: get\_student\_details

T->>DB: ORM query + serialise

DB-->>T: data

T-->>A: dict(profile)

A->>L: tool response

L-->>A: JSON answer

A-->>C: output

C-->>U: JSON payload

```

## 6 Error Handling & Edge Cases

\* \*\*Missing Student\*\* – Tools return `{ "error": "Student not found" }`; prompt instructs the LLM to surface this politely.

\* \*\*Ambiguous Names\*\* – LLM asks clarifying questions as guided by the prompt.

\* \*\*Incomplete Data\*\* – `null` fields plus a human-readable `message` key in the JSON.

\* \*\*Tool Failures\*\* – Wrapped in `try/except`, logged, and transformed into explanatory error dictionaries.

## 7 Design Decisions & Rationale

1. \*\*WebSockets over HTTP\*\* – Maintains a bi-directional channel for near real-time chat.

2. \*\*LangChain Agent Pattern\*\* – Simplifies multi-step tool invocation and scratch-pad management.

3. \*\*OpenAI Function Calling\*\* – Leverages the model’s native tool-calling capability for structured outputs.

4. \*\*Prompt-as-Spec\*\* – Prompt doubles as an executable \*contract\* and human-readable documentation.

5. \*\*Database-Centric Logic\*\* – Business rules (e.g. pass/fail) live in Django models, keeping the LLM stateless and data-agnostic.

## 8 Extensibility & Future Work

\* \*\*Additional Tools\*\* – Easily add `create\_student`, `update\_grade`, etc.

\* \*\*Caching Layer\*\* – Memoise frequent count queries for dashboard speed.

\* \*\*Role-based Access\*\* – Gate tool availability by user permissions.

\* \*\*Vector Search\*\* – Introduce semantic search on essay answers for richer analytics.

## 9 Glossary

\* \*\*Agent\*\* – A LangChain abstraction that wraps an LLM and a set of tools.

\* \*\*Tool\*\* – A callable (sync or async) that the LLM can invoke to fetch or mutate data.

\* \*\*Scratch-pad\*\* – Hidden messages that record intermediate tool calls.

\* \*\*LLM\*\* – Large Language Model (OpenAI GPT-4‐turbo / GPT-4o).

### © 2025 – Engineering Documentation Team

# Student Data Agent – Design & Architecture Documentation

## 1  Introduction

This document explains, **in prose only**, how the “Student Data Agent” works. It focuses on what the code does and why each design choice was made, without reproducing any source code. It is intended for engineers and product stakeholders who want to understand the architecture, data‑flow, and reasoning behind the system.

## 2  High‑Level Architecture

The solution turns a WebSocket message into a structured, tool‑augmented call to OpenAI and back to the client.

graph LR

UA([UserAgent –Browser]) -- text/json --> WS[[DjangoWebSocket Consumer]]

WS --> |parses & dispatches| AG[AgentExecutor]

AG --> |LLMPrompt & Tools| LLM[OpenAIChatModel]

LLM --> |function‑call style| DBTools[DatabaseToolsLayer]

DBTools --> |ORM queries| PG[(PostgreSQLDB)]

DBTools --> LLM

LLM --> AG

AG --> WS

WS --> UA

* **User Agent** – Sends natural‑language questions.
* **WebSocket Consumer** – Acts as the gateway, translating messages into agent invocations and relaying responses.
* **Agent Executor** – Manages prompt assembly, tool binding, scratch‑pad formatting, and conversation memory.
* **OpenAI Chat Model** – Interprets the prompt, decides when to call a tool, and generates JSON output.
* **Database Tools Layer** – Thin wrappers around Django ORM queries; exposed to the LLM as callable tools.
* **PostgreSQL** – Stores students, courses, grades, etc.

## 3  Prompt Strategy

1. **Role Definition** – The system message describes the agent’s twofold responsibility: (a) precise student‑data retrieval and (b) answering GK/Aptitude questions.
2. **Schema Enforcement** – The prompt mandates a **strict JSON schema** with keys such as student\_name, attendance, etc., guaranteeing machine‑readable replies.
3. **Tool Guidance** – Explicit instructions teach the LLM which tool answers which question, ensuring deterministic behaviour.
4. **Examples** – Few‑shot examples prime the model to mirror the desired structure and wording.
5. **Edge‑case Hints** – Clarifies how to react to missing data or ambiguous names.

### Why this matters

* Reduces hallucination by tightly scoping the agent’s domain.
* Off‑loads business logic (like field selection) to the LLM while keeping data access server‑side.

## 4  Tooling Strategy

|  |  |  |  |
| --- | --- | --- | --- |
| Tool (Logical Name) | Purpose | Typical Trigger Phrases | Notes |
| **get\_student\_details** | Fetch a complete student profile | “What is the GPA of…” | Serialises the Student object via DRF serializer. |
| **get\_student\_records** | Fetch a raw Student ORM instance | Internal helper when other tools need a base record. | Hidden from users. |
| **count\_total\_records** | Return counts for students, courses, etc. | “How many courses…”, dashboard stats | Uses a switch to target the right model. |
| **failed\_students** | List students whose performance status is Failed | “Show failed students” | Uses select\_related for efficiency. |
| **topper\_students\_list** | Top‑10 by GPA | “Top students” | Orders by -gpa and slices. |
| **get\_student\_session** | Pulls session‑specific records (attendance, grades…) | “Show internships for …” | Delegates to DRF serializers per relation. |

### Design Principles

* **Single Responsibility** – Each tool answers exactly one type of query.
* **Asynchronous Safety** – All synchronous ORM calls are wrapped with sync\_to\_async, preventing event‑loop blocking.
* **Consistent Interface** – Every tool returns plain Python structures (dict/list) that the LLM easily serialises.

## 5  Execution Flow (Sequence Diagram)

sequenceDiagram

participant U as User

participant C as ChatConsumer

participant A as AgentExecutor

participant L as OpenAI LLM

participant T as Tools Layer

participant DB as Database

U->>C: "What is the attendance of Alexis?"

C->>A: invoke agent

A->>L: system / user / scratchpad messages

L-->>A: function‑call get\_student\_details(name="Alexis")

A->>T: get\_student\_details

T->>DB: ORM query + serialise

DB-->>T: data

T-->>A: dict(profile)

A->>L: tool response

L-->>A: JSON answer

A-->>C: output

C-->>U: JSON payload

## 6  Error Handling & Edge Cases

* **Missing Student** – Tools return { "error": "Student not found" }; prompt instructs the LLM to surface this politely.
* **Ambiguous Names** – LLM asks clarifying questions as guided by the prompt.
* **Incomplete Data** – null fields plus a human‑readable message key in the JSON.
* **Tool Failures** – Wrapped in try/except, logged, and transformed into explanatory error dictionaries.

## 7  Design Decisions & Rationale

1. **WebSockets over HTTP** – Maintains a bi‑directional channel for near real‑time chat.
2. **LangChain Agent Pattern** – Simplifies multi‑step tool invocation and scratch‑pad management.
3. **OpenAI Function Calling** – Leverages the model’s native tool‑calling capability for structured outputs.
4. **Prompt‑as‑Spec** – Prompt doubles as an executable contract and human‑readable documentation.
5. **Database‑Centric Logic** – Business rules (e.g. pass/fail) live in Django models, keeping the LLM stateless and data‑agnostic.

## 8  Extensibility & Future Work

* **Additional Tools** – Easily add create\_student, update\_grade, etc.
* **Caching Layer** – Memoise frequent count queries for dashboard speed.
* **Role‑based Access** – Gate tool availability by user permissions.
* **Vector Search** – Introduce semantic search on essay answers for richer analytics.

## 9  Glossary

* **Agent** – A LangChain abstraction that wraps an LLM and a set of tools.
* **Tool** – A callable (sync or async) that the LLM can invoke to fetch or mutate data.
* **Scratch‑pad** – Hidden messages that record intermediate tool calls.
* **LLM** – Large Language Model (OpenAI GPT‑4‐turbo / GPT‑4o).

### © 2025 – Engineering Documentation Team