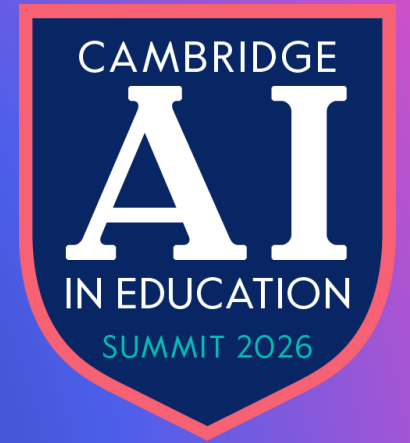




Rethinking programming education in the age of AI agents

Dr Marcos Barreto, Anuhya Thota, Mengning Chen

Department of Statistics – London School of Economics and Political Science (LSE)

m.e.barreto@lse.ac.uk

12/10/2026

Organised by



Summit partners



Funding and support note

- Funding



- Support



- Testers: Steven Williams (AI and Education Specialist, LSE), Alice Lockwood (BSc Politics and Data Science), Oscar Wedmore (BSc Mathematics with Data Science), Anastasiia Prokhorova (MSc Data Science)

Motivating question

- What's left to programmers if AI can generate high-quality, complex code?

The **key challenges** for students remain the same:

- Formulating problem requirements in sufficient detail;
- Testing/recognising problems in the code;
- Refining code; and
- Designing an entire system.

(Adapted from Michael Hansen. **Top 10 things I learned from ChatGPT**. AMSTASTS NEWS. September 2023. https://magazine.amstats.org/blog/2023/09/01/hansen_chatgpt)

Viewpoint

The End of Programming

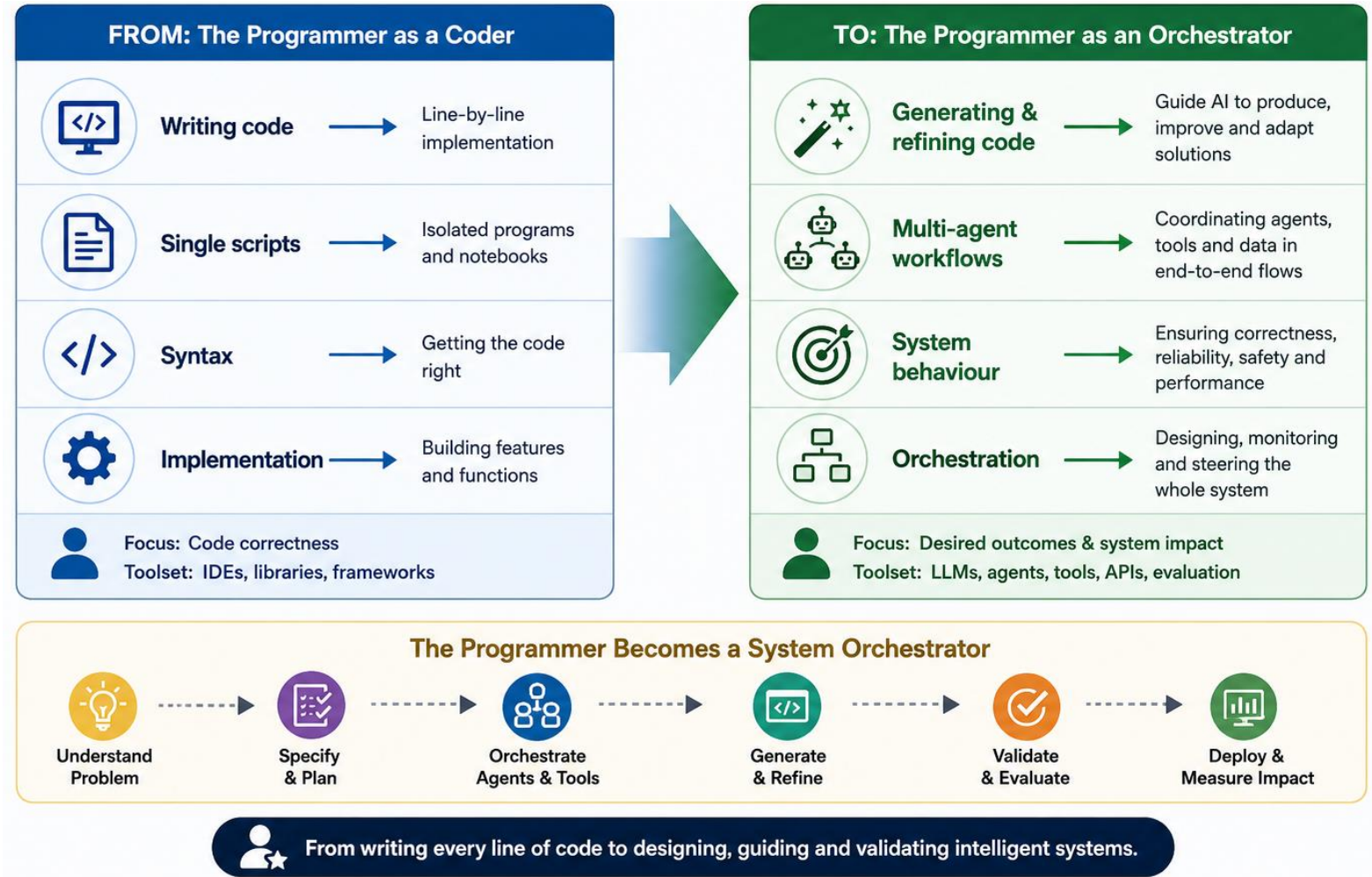
The end of classical computer science is coming, and most of us are dinosaurs waiting for the meteor to hit.

'I believe the conventional idea of “**writing a program**” is headed for extinction, and indeed, for all but very specialized applications, **most software**, as we know it, **will be replaced by AI systems** that are *trained* rather than *programmed*.'



Proposed work

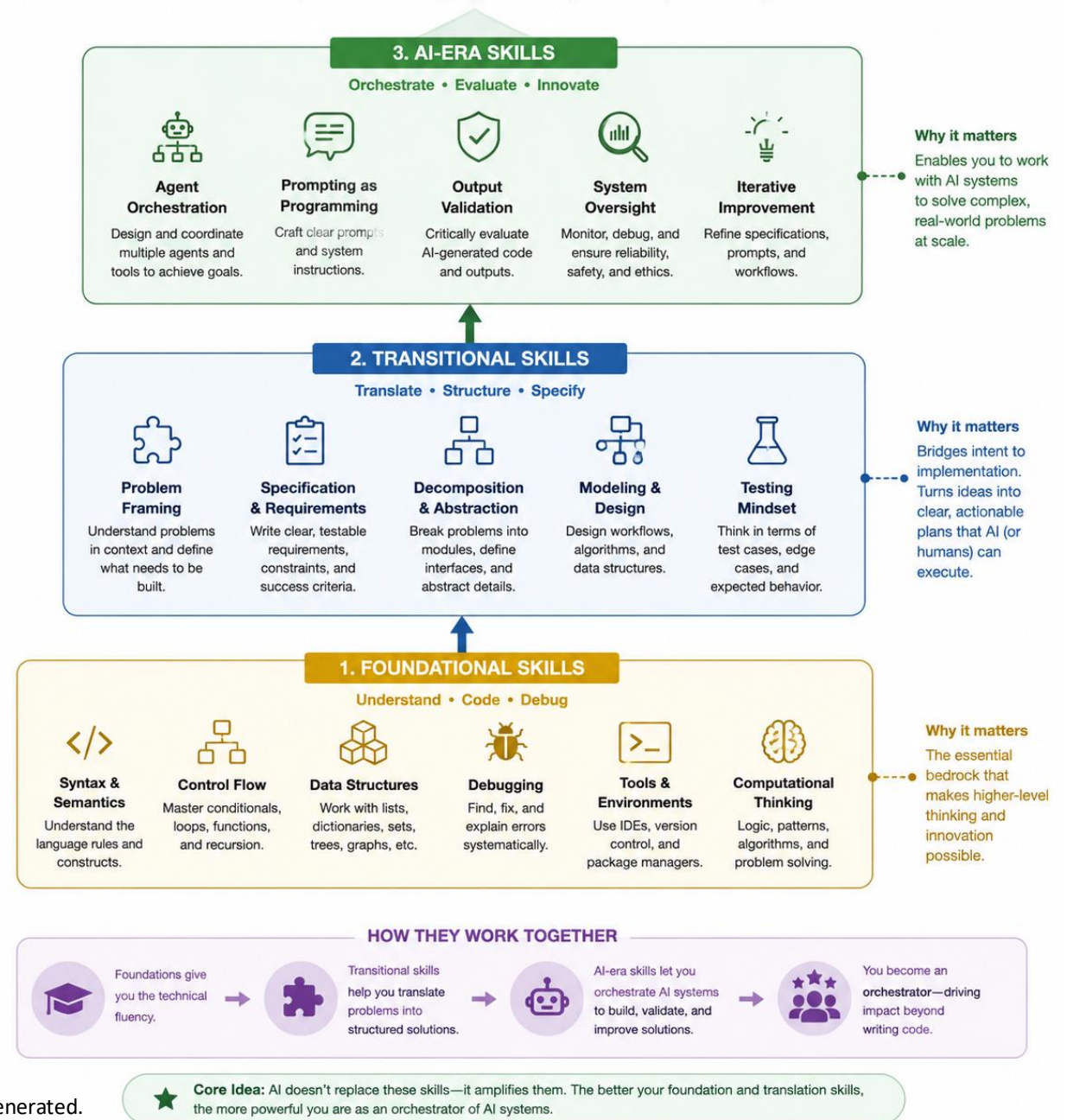
- Revise the key challenges and rethink programming education towards them.



Generative AI generated.

Proposed work *(cont.)*

- Define baseline **AI-assisted programming skills**:
- Foundational (non-negotiable)
- Transitional
- Agentic AI



Generative AI generated.

Evidence from literature*

- **AI as tutor**
 - Teaching assistants providing feedback
- **AI as collaborator**
 - Pair programming/debugging
 - Iterative refinement
- **AI as object of reasoning**
 - Students debugging AI-generated code, implementation plans, and unit tests

* Academic publications from 2024-2026 on “AI agents for programming and data science education”.

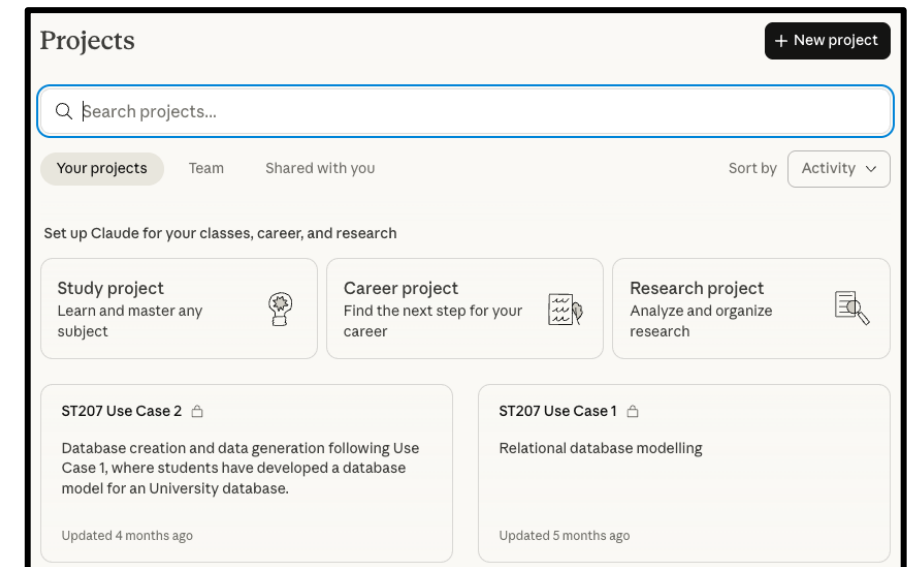
Early experimentation

- **2023-2024**



- 1st LSE-funded project on GenAI in education
- 7 courses, 4 departments, 220 students
- AI-supported exercises, including programming

- **2025**



Current approach

- Teaching resources for typical data science tasks
 - Database modelling and programming
 - Supervised and unsupervised machine learning
- AI agents as collaborators under students' guidance
 - Recurring cycle: specify => guide => generate => inspect => critique => refine
- Experimentation with Google tools and different AI-exposure levels
 - Antigravity 2.0, Antigravity SDK, Colab

Example: database modelling and programming

- Structure and expected artefacts

[text] denotes who is responsible for producing the artefacts

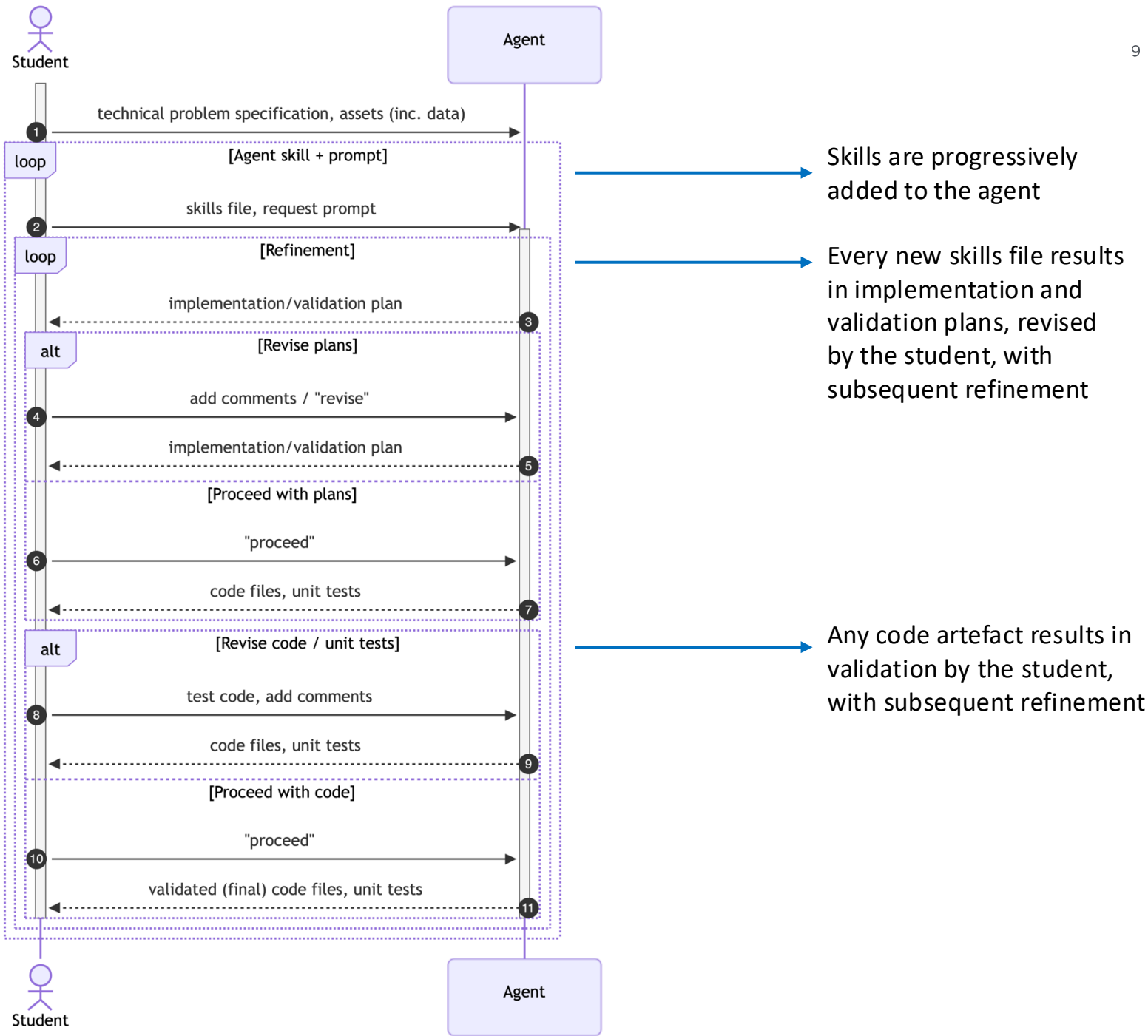
```

CompanyDB/
├── .agents/
│   └── skills/
│       ├── companydb_modelling/
│       │   └── SKILL.md
│       ├── companydb_data_insertion/
│       │   └── SKILL.md
│       └── companydb_querying/
│           └── SKILL.md
├── assets/
│   ├── data
│   └── er_diagram.png
├── db/
├── references/
│   └── technical_specification.md
├── code/
│   ├── Python scripts
│   └── Jupyter notebooks
├── docs/
│   ├── implementation_plan.md
│   ├── validation_plan.md
│   ├── task_list.md
│   ├── walkthrough.md
│   └── report.pdf
├── README.md
└── database_description.md

# [instructor/student] Project repository
# Standard folder for agent skills
# Agent skills
# [student] Translates specs to schema and verifies setup
# [student] Ingests test data and verifies rules
# [student] Writes SELECT queries and cross-checks with Python
# Project assets
# [instructor/student] Raw or synthetic data
# [student] ER database model
# [student/AI agent] Output folder for SQLite database files
# Technical specification documents
# [student] Source of truth for database schema and constraints
# [AI agent] Generated code
# [AI agent] Expected one script per skill
# [AI agent] Consolidated Python code (Google Colab compatible)
# Process-based documents
# [AI agent] Project implementation plan
# [AI agent] Project validation plan
# [AI agent] Agent's task list
# [AI agent] Project stepwise report
# [student] Reflective report
# [instructor] Exercise specification
# [instructor] Database scenario & analytical questions
    
```

Example: database modelling and programming

- Student-agent collaboration



Example: database modelling and programming

- Skills file

name	companydb_modelling
description	Reviews an E-R model against a technical specification and generates a SQLite schema implementation plan, scripts, and verification tests. Use this skill when the user asks for help creating or generating a database.

Goal

Validate the database E-R model stored as `assets/er_diagram.png` against `references/technical_specification.md` and produce a schema implementation plan.

Instructions

- 1. Read Specifications:** Read `references/technical_specification.md` at the project root. This is the source of truth for entities, attributes, relationships, cardinalities, and business rules.
- 2. Review E-R Diagram:** Review `assets/er_diagram.png` (or `.mmd` source) at the project root. If no diagram is found, STOP, and ask the user to produce a valid diagram matching the technical specification document.
- 3. Compare:** Compare the diagram against the technical specification. Identify any missing or extra entities, attributes, relationships, cardinalities, or specialisations. Do NOT add anything not explicitly defined.
- 4. Mismatches:** If there are mismatches, STOP. Report them and wait for user input before proceeding. Do not silently modify the model.
- 5. Implementation Plan:** Generate an implementation plan stating how the E-R model/diagram and the technical specification will be mapped into a database schema. Present the implementation plan to the user and wait for their approval before moving forward. The implementation plan may contemplate the following mappings:
 - Entities to tables
 - 1:N relationships to foreign keys
 - N:M relationships to junction tables
 - Subtypes (specialisation) to tables (explain the strategy: single table vs. one per subtype)
 - Any other mapping operations, following the E-R model/diagram and technical specification document.
- 6. SQLite Schema Scripts:** Once the implementation plan is approved, generate Python scripts (not raw SQL files) to create the schema (primary keys, foreign keys, CHECK constraints). Target database is SQLite. Check if files exist at the target path first; do not overwrite without user confirmation. Place all generated Python scripts in the `code` folder, and store the SQLite database file in `data`.
- 7. Verification Script:** Generate a validation plan to verify the schema against the technical specification. Present the plan to the user for approval. Once approved, create a Python script for the following statements:
 - `SELECT name FROM sqlite_master WHERE type='table'`
 - `PRAGMA foreign_key_list(table)`
 - `PRAGMA foreign_keys` (to verify enforcement of foreign keys)the schema and re-run.

Constraints

- Enforce business rules from `references/technical_specification.md` only.
- For single-column/single-row constraints, use SQLite `CHECK` constraints.
- For multi-row or multi-table constraints, delegate to application logic or database triggers; call these out in the plan.
- If the diagram and technical specification disagree, clarify with the user.

Run the `companydb\_modelling` skill to create the database schema based on the referenced workspace artefacts. Do not go beyond the instructions in the skills file; more specifically, do not create any code to load data into the database or implement any query. This is the first step in the database modelling exercise where we will only create an empty database matching the provided description and schema. Before generating the database schema and any code files, generate implementation and validation plans and wait for my review and approval.

Request prompt

name

companydb_modelling

description

Reviews an E-R model against a technical specification and generates a SQLite schema implementation plan, scripts, and verification tests. Use this skill when the user asks for help creating or generating a database.

Goal

Validate the database E-R model stored as `assets/er_diagram.png` against `references/technical_specification.md` and produce a schema implementation plan.

Instructions

1. Read Specifications:

Read `references/technical_specification.md` at the project root. This is the source of truth for entities, attributes, relationships, cardinalities, and business rules.

2. Review E-R Diagram:

Review `assets/er_diagram.png` (or `.rmd` source) at the project root. If no diagram is found, STOP, and ask the user to produce a valid diagram matching the technical specification document.

3. Compare:

Compare the diagram against the technical specification. Identify any missing or extra entities, attributes, relationships, cardinalities, or specialisations. Do NOT add anything not explicitly defined.

4. Mismatches:

If there are mismatches, STOP. Report them and wait for user input before proceeding. Do not silently modify the model.

5. Implementation Plan:

Generate an implementation plan stating how the E-R model/diagram and the technical specification will be mapped into a database schema. Present the implementation plan to the user and wait for their approval before moving forward. The implementation plan may contemplate the following mappings:

- Entities to tables
- 1:N relationships to foreign keys
- N:M relationships to junction tables
- Subtypes (specialisation) to tables (explain the strategy: single table vs. one per subtype)
- Any other mapping operations, following the E-R model/diagram and technical specification document.

6. SQLite Schema Scripts:

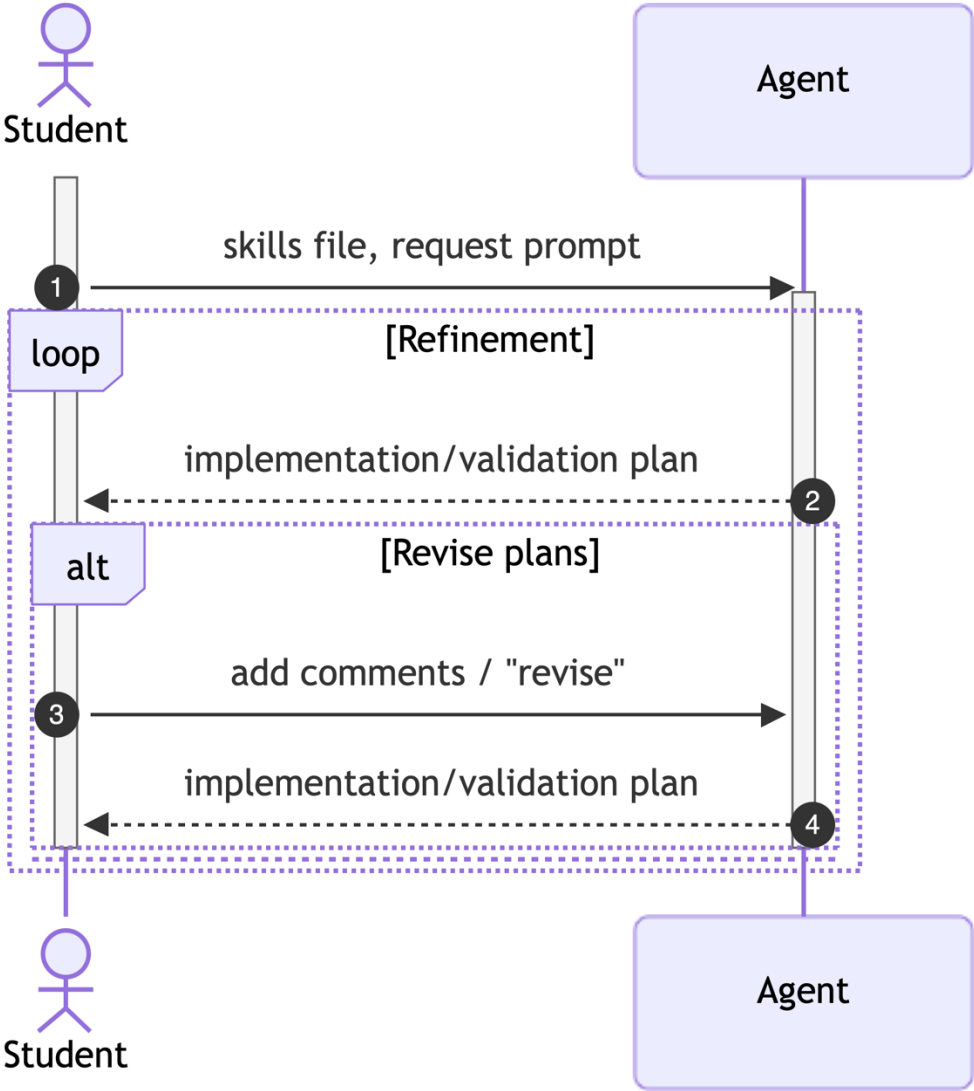
Once the implementation plan is approved, generate Python scripts (not raw SQL files) to create the schema (primary keys, foreign keys, CHECK constraints). Target database is SQLite. Check if files exist at the target path first; do not overwrite without user confirmation. Place all generated Python scripts in the `code` folder, and store the SQLite database file in a `db` folder.

7. Verification Script:

Generate a validation plan to inspect the created schema in terms of tables, attributes, data types, primary and foreign keys. Present the validation plan to the user and wait for their comments and approval. Once approved, create a Python script for that and save it to the `code` folder. You can also use the following statements:

- `SELECT name FROM sqlite_master WHERE type='table'`
- `PRAGMA foreign_key_list(table)`
- `PRAGMA foreign_keys` (to verify enforcement is ON) Verify the empty schema. If the checks fail, fix the schema and re-run.

Skills file



Implementation & Validation Plan: CompanyDB Database Schema

This plan describes the mapping of the Company DB E-R model (located in `assets/er_diagram.png`) and reference specifications (in `references/technical_specification.md`) to a SQLite database schema. It also outlines the validation plan to verify the correctness of the generated empty database.

1. E-R Diagram vs. Technical Specification Review

A thorough comparison has been performed between the E-R diagram (`assets/er_diagram.png`), the database description (`database_description.md`), and the technical specifications (`references/technical_specification.md`):

1. Entities & Attributes Match:

- EMPLOYEE** : Matches perfectly. Composite attribute (**Name**) composed of **Fname** , **Minit** , **Lname** is correctly flattened into separate columns in the specifications. **Ssn** is correctly marked as the primary key.
- DEPARTMENT** : Matches perfectly. **Dnumber** is the primary key. **Dname** is unique. **Locations** is a multi-valued attribute (represented by double oval), which maps to a separate table `dept_locations` in the specifications. **Number_Employees** is a derived attribute (dashed oval) and is omitted from physical persistence as per standard practices.
- PROJECT** : Matches perfectly. **Pnumber** is the primary key. **Pname** is unique. **Plocation** is simple.
- DEPENDENT** : Matches perfectly. Represented as a weak entity (double rectangle) with **Identifying relationship**. This maps to a con

• **Manages** (EMPLOYEE to DEPARTMENT): 1:1 relationship. An employee manages 0 or 1 department (**Mgr_ssn** is nullable).

Mgr_ssn is an attribute of DEPARTMENT acting as a foreign key to the employee's Ssn who manages the department. All departments must have a manager, so this attribute cannot be null.

Cancel Add Comment

2. Relationships & Cardinalities

- Works_for** (EMPL belongs to exactly
- Manages** (EMPLOYEE to DEPARTMENT): 1:1 relationship. An employee manages 0 or 1 department (**Mgr_ssn** is nullable).
- Supervision** (EMPLOYEE to EMPLOYEE): Recursive 1:N relationship. An employee has 0 or 1 supervisor (**Super_ssn** is nullable).
- Controls** (DEPARTMENT to PROJECT): 1:N relationship. Each project is controlled by exactly one department (**Dnum** is **NOT NULL**).
- Works_on** (EMPLOYEE to PROJECT): N:M relationship. Maps to a junction table `works_on` with composite primary key (**Essn**, **Pno**).
- Dependents_of** (EMPLOYEE to DEPENDENT): Weak entity relationship. Maps to foreign key **Essn** in `dependent` table, which is also part of the primary key.

Mismatches/Issues identified: None. The physical specification in `references/technical_specification.md` is fully consistent with the E-R diagram.

I have completed all the tasks according to the updated database modelling instructions and your feedback.

Here is a summary of the accomplishments:

1. Documentation Written:

- `docs/implementation_plan.md`: Details the design mappings of entities, attributes, relationships, and recursive links to physical tables.
- `docs/validation_plan.md`: Describes the structural testing and how to run verification tests programmatically in Google Colab.
- `docs/task_list.md`: Tracks completed and remaining work.

2. Schema Definition Script:

- Written to `code/create_schema.py`.
- This script constructs `db/company.db` and maps the correct tables with your requested adjustments:
 - `Mgr_ssn` is enforced as `NOT NULL`.
 - `works_on` uses `Ssn` instead of `Essn`.
 - `Relationship` in the `dependent` table is typed as `VARCHAR(30)`.

3. Validation Suite:

- Written to `code/verify_schema.py`.
- Uses only Python's standard `unittest` and `sqlite3` libraries and is fully prepared to be imported/called directly inside Google Colab cells without executing command-line scripts.
- Checks table structures, data types, nullabilities, and foreign key relations.

4. Execution and Validation:

- Ran the schema generation script successfully.
- Ran the verification test suite, which executed and passed all 9 test cases successfully.

You can view the detailed results in the [Walkthrough Report](#). The empty database schema is now fully initialized and validated, ready for the next steps:

Student

Agent

'proceed with plans'

code files, unit tests

[Revise code / unit tests]

st code, add comments

code files, unit tests

[Proceed with code]

CompanyDB_v1 > code > create_schema.py (all agent edits)

```
1 import os
2 import sqlite3
3
4 def create_database_schema(db_path="db/company.db"):
5     # Ensure the directory exists
6     db_dir = os.path.dirname(db_path)
7     if db_dir and not os.path.exists(db_dir):
8         os.makedirs(db_dir)
9
10    print(f"Connecting to database at: {db_path}")
11    conn = sqlite3.connect(db_path)
12    cursor = conn.cursor()
13
14    # Temporarily disable foreign keys to safely drop and recreate tables
15    cursor.execute("PRAGMA foreign_keys = OFF;")
16
17    print("Dropping existing tables")
18    # Temporarily disable foreign keys to safely drop and recreate tables
19    cursor.execute("PRAGMA foreign_keys = OFF;")
20    for table in tables:
21        cursor.execute(f"DROP TABLE {table};")
22    print("Tables dropped")
23
24    # 1. Create Department table (referencing employee)
25    cursor.execute("""
26    CREATE TABLE department (
27        Dname VARCHAR(30) NOT NULL UNIQUE,
28        Dnumber INT PRIMARY KEY,
29        Mgr_ssn CHAR(11) NOT NULL,
30        Mgr_start_date DATE,
31        FOREIGN KEY (Mgr_ssn) REFERENCES employee(Ssn) ON DELETE RESTRICT ON UPDATE CASCADE
32    );
33    """)
```

Code file

Walkthrough

Final walkthrough process-based report for CompanyDB database engineering.

5 files changed +404 -0

- `create_schema.py` /Users/m.e.barreto/Tools/Google_AGY/CompanyDB_v1/code
- `verify_schema.py` /Users/m.e.barreto/Tools/Google_AGY/CompanyDB_v1/code
- `implementation_plan.md` /Users/m.e.barreto/Tools/Google_AGY/CompanyDB_v1/docs
- `task_list.md` /Users/m.e.barreto/Tools/Google_AGY/CompanyDB_v1/docs
- `validation_plan.md` /Users/m.e.barreto/Tools/Google_AGY/CompanyDB_v1/docs

Current approach *(cont.)*

- Human agency + guardrails

Implementation & Validation Plan: Collaborative SQL Querying

This plan outlines the approach to collaboratively write, validate, and programmatically cross-check SQL queries for the CompanyDB analytical questions, in strict adherence to the `companydb_querying` skill.

1. Collaborative Query Writing Strategy

As mandated by the `companydb_querying` skill, we will treat each query as a collaborative process. For every analytical

- 1. Identify the Objective:** State the question clearly.
- 2. Interactive Design:** We will ask you to guide us on:
 - Which tables need to be queried (`FROM` clause).
 - How to structure any necessary `JOIN` operations
 - Any conditions or filtering rules to apply (`WHERE` clause).
 - Which columns to project (`SELECT` clause).
- 3. Query Construction:** We will synthesize the read-only query based on your inputs and our verified SQLite schema.
- 4. Interactive Log:** We will capture this collaborative design and implementation plan stored in the `docs/` folder.

2. Programmatic Verification Strategy

To guarantee the correctness of each generated SQL query and avoid hallucination, we will implement a cross-checking strategy using `code/verify_queries.py`. For each query, this script will:

- 1. Execute the SQL Query:** Run the proposed SQL query against the database and retrieve the result set.
- 2. Independent Python Logic Calculation:** Implement equivalent logic in Python (using lists, dicts, or other standard data structures) to perform the equivalent filtering/aggregating/joining and compare the result to the expected result.
- 3. Assert Equality:** Verify that the SQL result set matches the Python result set exactly (structure and values).
- 4. Output Report:** Save a verification report in `docs/query_verification_report.md`.

3. Immediate Action: Waiting for Your Analytical Questions

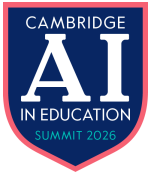
Since the current workspace files (such as `database_description.md` and `README.md`) contain instructions but do **not** enumerate the specific analytical questions to be answered, we need you to provide them.

To help you formulate the queries and guide the process, here is a quick reference of our validated schema:

- **employee**: `Fname`, `Minit`, `Lname`, `Ssn` (PK), `Bdate`, `Address`, `Gender`, `Salary`, `Super_ssn` (FK to employee), `Dno` (FK to department).
- **department**: `Dname` (Unique), `Dnumber` (PK), `Mgr_ssn` (FK to employee, NOT NULL), `Mgr_start_date`.
- **dept_locations**: `Dnumber` (FK to department), `Dlocation` (PK, composite).
- **project**: `Pname` (Unique), `Pnumber` (PK), `Plocation`, `Dnum` (FK to department, NOT NULL).
- **works_on**: `Ssn` (FK to employee), `Pno` (FK to project), `Hours` (PK, composite).
- **dependent**: `Essn` (FK to employee), `Dependent_name` (PK, composite), `Gender`, `Bdate`, `Relationship`.

Ready for Your Instructions

Please provide the analytical questions you would like us to solve. For each question, feel free to give us your thoughts on the tables, joins, filters, and projections, or simply list the questions and we will work through them one by one!



Current approach *(cont.)*

- Human agency + guardrails

I see the updated section in `database_description.md` now! Excellent.

Here are the three analytical questions we need to answer. To make this a truly collaborative effort, let's break down each question.

For each question below, I have provided a quick hint or visual of the tables involved, and a prompt to guide me:

- Which tables we should query (`FROM` clause)?
- Which conditions to apply (`WHERE` or `ON` clause)?
- How we should structure the `JOIN` operations (e.g. `INNER JOIN`, `LEFT JOIN`)?
- Which columns we should project in the output (`SELECT` clause)?

Question 1: Group projects per department and list all employees working on them.

- Objective:** Show the departmental organization of projects, along with the employee count.
- Key entities involved:** Departments, Projects, Employees, and the assignments between them.
- Tables to think about:** `department`, `project`, `employee`, and `works_on`.

NOTE

How would you structure the `FROM` and `JOIN` clauses to connect Departments with Projects and Employees? Which fields should we project to make the list clear?

For question 1, join department and project tables to retrieve the department name, all project names, and the employee count. Let's do question 1 first and add more questions later.

Thought for 11s >

This is excellent guidance! Based on your instructions, I will structure the SQL query as follows:

- Join `department` and `project` by the `department_id` foreign key.
- Retrieve employees working on each project via the `works_on` table.
- Retrieve the department name, project name, and the count of employees per project.

Since we want to "group projects per department" and list all employees working on them, we have two main ways to structure the query:

Option A: Tabular Layout (Grouping by Department)

This structure groups the results by department and lists all projects within each department, along with the employee count for each project. It's the standard way to present this data.

```
sql
SELECT
  d.Dname AS Department_Name,
  p.Pname AS Project_Name,
  COUNT(w.Ssn) AS Employee_Count
FROM department d
JOIN project p ON d.Dnumber = p.Dnum
LEFT JOIN works_on w ON p.Pnumber = w.Pno
GROUP BY d.Dname, p.Pname
ORDER BY d.Dname, p.Pname;
```

I have successfully implemented, tested, and programmatically verified **Question 1!**

1. Results for Question 1 (Projects per Department with Employee Count)

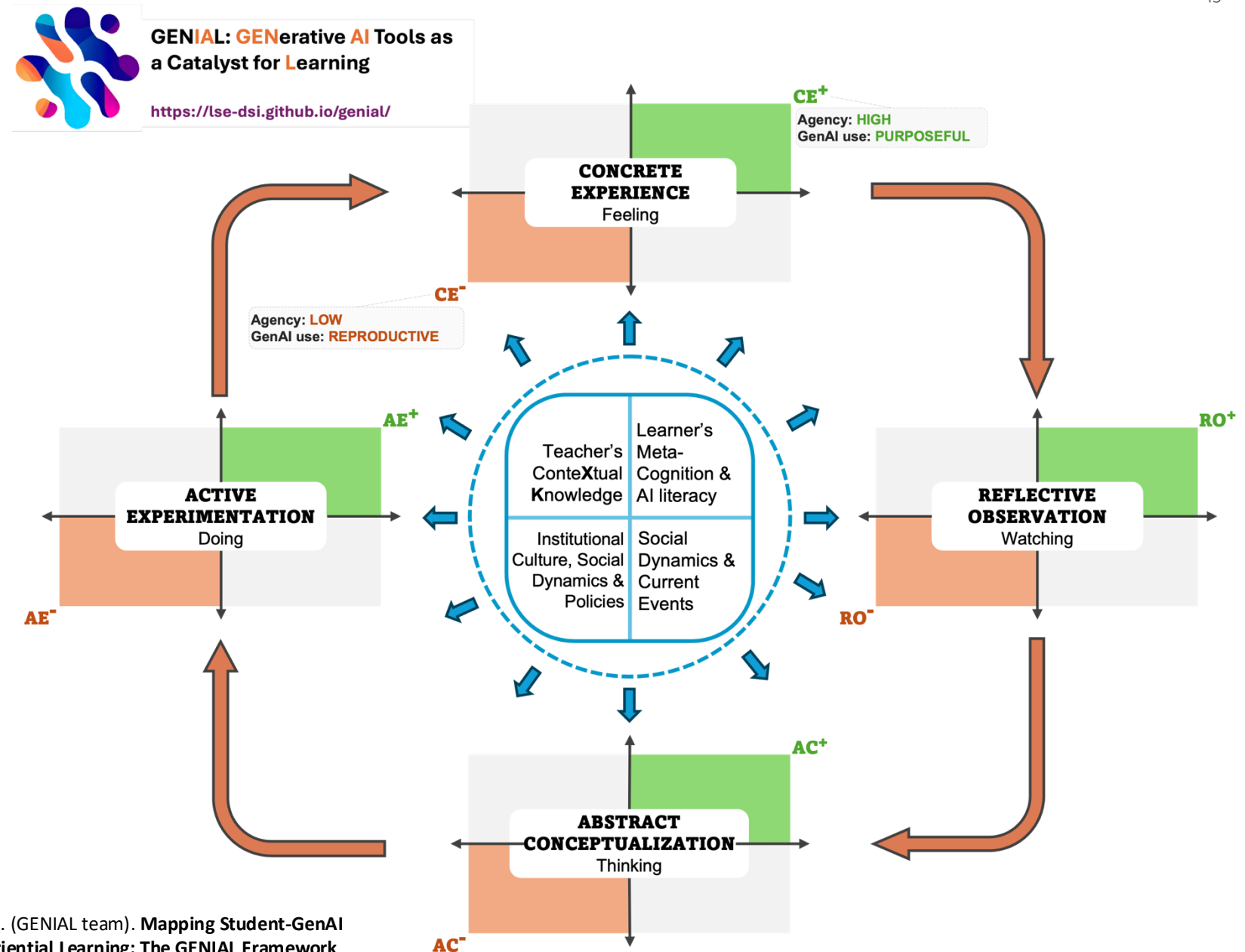
The SQL query was run against the database and cross-checked against independent Python aggregation logic. Both matches were 100% identical. Here is the verified database output:

Department Name	Project Name	Employee Count
Administration	Computerization	3
Administration	Newbenefits	3
Hardware	InkjetPrinters	8
Hardware	LaserPrinters	3
Headquarters	Reorganization	3
Research	ProductX	2
Research	ProductY	3
Research	ProductZ	2
Software	DatabaseSystems	8
Software	Middleware	4
Software	OperatingSystems	9

The SQL query and design notes have been captured in `docs/query_implementation_plan.md`, and the test suite has been saved in `code/verify_queries.py`.

Analysis of students' reflective reports

- **GENIAL framework** applied to student- and AI-generated outputs
- Analysis of skills files against reference implementation



Marcos Barreto et al. (GENIAL team). **Mapping Student-GenAI Interactions onto Experiential Learning: The GENIAL Framework.** October 2025. Available at SSRN: <https://ssrn.com/abstract=5674422>

Example resources (teaching)

- **Database modelling**

- Company (lecture / formative)
- Private Equity (Antigravity 2.0)
- e-Commerce (Antigravity SDK)

- **Machine learning pipelines**

- (Regression) Prediction of loan interest rate (Antigravity 2.0)
- (Classification) Image classification (Antigravity SDK)
- (Unsupervised) Customer segmentation / Market basket analysis
(under development)

Feedback and results

- **Being piloting by students in Autumn Term (AT)**

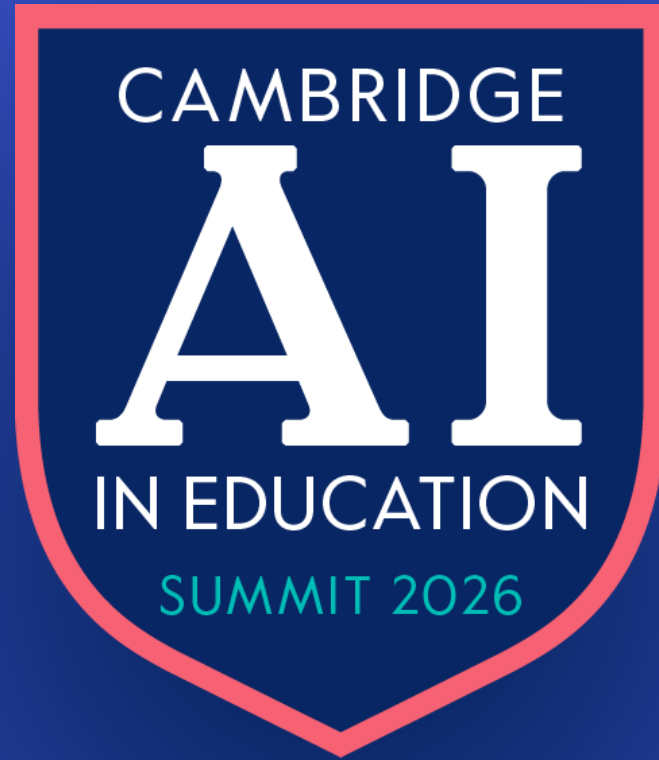
- Databases: 63 students, 6-23 October
- Machine learning: 57 students, late AT

- **Testers' feedback**

- Agent's ability to cross-checking the E-R model against the technical specification document and solve mismatches
- Necessary guardrails to prevent the agent to hallucinate (beyond the skills file) and/or give full answers without student approval
- Stress test of generated code and unit tests
- Clarification of starting point (problem specification, assets, expected outputs)

Ongoing work and collaboration

- Piloting some teaching resources in Autumn Term (AT)
- Implementing unsupervised learning examples
- Enhancing the AI usage declaration and the process and resources for student reports
- Considering a coding-focused Generative AI Declaration Framework
- **Contact:** Dr Marcos Barreto (m.e.barreto@lse.ac.uk)



Organised by



Summit partners



aisummit.cambridge.org

© Cambridge University Press & Assessment 2026