applied sciences

MDPI

Article

Teaching SQL with an Agentic LLM: A Fifteen-Category Hallucination Taxonomy and Mitigation Strategy, Illustrated with Claude Sonnet 5

Paolino Di Felice

Department of Industrial and Information Engineering and Economics, University of L'Aquila, 67100 L'Aquila, Italy; paolino.difelice@univaq.it

Abstract

Students increasingly use large language models (LLMs) to translate natural-language requests into SQL, yet these models hallucinate: they return queries, numbers, explanations or assurances unsupported by the schema, data, or request, often in a form that executes without error. This paper proposes a teaching instrument for a first course on relational databases, designed to let students experience, recognize and prevent such failures; its contribution is conceptual and illustrative. This paper proposes a teaching instrument for a first course on relational databases that lets students experience, recognize, and prevent such failures. Its core is a taxonomy of fifteen hallucination categories that is coarse enough to serve as a classroom checklist, identified through an expert-driven process (the text-to-SQL literature and 27 years of teaching experience) and illustrated on a five-table PostgreSQL schema through constructed baseline outputs that the instructor can use to open a classroom discussion of each failure. Each category is paired with a two-layer mitigation: a context pack of standing rules and seven reference files, stored once in a Claude Project, and an agentic execute–observe–correct loop in which the assistant runs rule-prescribed checks against a live database before answering. Each countermeasure is assigned to the instructor, the loop, or the user. Nine categories have a counterpart in prior text-to-SQL taxonomies, two lack a dedicated one, and four concern the answer returned to the user rather than the query, thereby escaping static benchmark-based evaluation. The workflow is free for students. The effectiveness and generalizability of the mitigation have not yet been systematically validated; frequency and classroom effectiveness are left to future work.

Keywords: database education; SQL learning; text-to-SQL; large language models; LLM hallucination; error taxonomy; agentic AI; execution-based verification; critical AI literacy; equity of access

Check for updates

Academic Editors: Arkadiusz Gola, Ascensión López-Vargas and Ángel García-Beltrán

Received: 20 September 2026
Revised: 4 November 2026
Accepted: 19 November 2026
Published: 23 November 2026

Copyright: © 2026 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

The context motivating this study is the rapid uptake of agentic LLM-based tools across university curricula. Adoption of generative and agentic AI in higher education has grown from isolated experimentation into a distinct research strand: empirical and stakeholder studies examine how staff and students adopt and regulate these tools [1–4], bibliometric reviews chart the expansion of the field across world regions and sub-topics [5,6], and reviews of LLM-based chatbots document a broadening set of institutional deployments, from personalized tutoring pilots to enterprise-scale partnerships between universities and

AI vendors [7]. Database systems education has its own emerging body of work on LLM integration, spanning dedicated workshop venues for data systems education research [8], LLM-driven feedback on conceptual (entity relationship) design exercises [9], capstone project redesigns of database courses around an LLM-based tool [10], and automated personalized feedback for SQL exam grading [11]. Closest to the present paper is prior work on using LLMs to generate natural-language translations and explanations of SQL queries for classroom use [12,13], to grade student-submitted SQL and give feedback on it [14], and to design instructor-facing tips for bringing AI tools into SQL exercises [15]. This paper is concerned with one aspect of this last setting: the hallucinations that an agentic LLM can produce when it translates a natural-language request into SQL, and the way a database course can turn these hallucinations into teaching material. Following the text-to-SQL literature [16,17], which counts invented functions and wrong clause or operator logic among hallucinations, we use the term *hallucination* for any output that the assistant presents as a correct answer to the request, either explicitly or implicitly, but that is not in fact supported by the declared schema, data, request, semantics of the target dialect, or established knowledge, whether that output is a query, a number, an explanation or an assurance. The above definition is deliberately broader than the generated SQL: what a student receives from an assistant is an answer, of which the query is only one part.

1.1. Motivation

Four considerations motivate the work:

- The centrality of SQL. SQL has been described as the backbone of modern database systems ever since its introduction in the 1970s [18], and it remains the standard language for querying relational data in academic and professional settings alike [19]. Thus, learning to formulate correct SQL queries is not one skill among many in a database course; it is the skill that most directly carries over into professional practice.
- The difficulty of learning SQL. Formulating correct queries is hard for novices; large-scale analyses of student submissions have catalogued recurring syntax, semantic, and logical errors [20], while think-aloud studies have traced many of these instances to underlying misconceptions rooted in prior course knowledge, over-generalization, natural-language interference, or an incomplete relational mental model [21]. Experienced instructors have corroborated and extended these findings [22]. We have observed the same difficulties firsthand over 27 years of teaching database systems to first-year Master's students in Management Engineering. The same errors recur cohort after cohort, especially around joins, grouping and aggregation, NULL handling, and the translation of informally stated business requirements into relational logic. Several of these recurring errors reappear among the hallucination categories proposed in this paper. An LLM can reproduce the very errors that instruction is meant to eliminate, fluently and with apparent authority.
- The timeliness of integrating LLM-based tools into SQL teaching. Agentic LLM-based assistants are now part of how students study and how professionals write SQL, and are often proposed to fill precisely the gap that the difficulty of SQL leaves open [23–25]. Database education research has started to integrate these tools into feedback, grading, project work, and exercise design [8–15]. For an instructor, the question is no longer whether students will use these tools but how traditional teaching should be combined with them so that they support learning rather than replacing it.
- The urgency of educating students in the appropriate use of these tools. The risks of LLMs in education are well documented: hallucination, overreliance, opacity, and the erosion of independent problem-solving skills [3,26–28]. In adjacent code generation

settings, most students accepted deliberately insecure LLM-generated code even after security coursework and while stating that they distrusted the assistant [29]; classroom experience with database courses likewise shows that prior knowledge of the subject is what allows students to notice when an assistant is wrong [30]. An instructor who presents AI-generated SQL without verification will risk embedding misconceptions that are difficult to unlearn. This concern is especially acute at the Master's level, where students carry these patterns into professional or research practice. Hallucination is especially insidious in text-to-SQL, because a hallucinated query is often syntactically valid, executes without error, and returns a plausible result; therefore, students need to be trained to recognize, check, and prevent hallucinations before relying on such tools unsupervised. Letting students gain first-hand experience with the many forms of hallucination that arise when a relational database is queried in natural language is the motivation from which this work stems.

1.2. Hallucination in Text-to-SQL and Open Gaps

Hallucination in text-to-SQL differs from hallucination in open-domain text generation in one important respect: a generated query is executable, so the database itself can act as an external oracle. The oracle, however, answers only the questions that are put to it. The engine reports an error when a query refers to a table, column, or function that does not exist, but most hallucinated queries execute without error and return a plausible result; in a recent annotation of 3968 incorrect LLM-generated queries, about 78% of the errors were of this silent kind [31]. Detecting such errors requires checks that compare the result with something else, such as a probe of the stored values, an independent count of the same set, or a reference query with a known result. The model cannot be relied upon to judge its own output, as benchmark studies show that LLMs asked whether a query is correct miss a large share of errors [31,32]. This motivates the mitigation philosophy adopted here: each generated query is treated as a hypothesis to be executed, checked, and where necessary, revised before it is presented; the checks to be run are prescribed in advance rather than left to the model's judgment.

Against this background, three gaps remain open in the literature on text-to-SQL hallucinations:

Granularity. The schema-based/logic-based dichotomy of Qu et al. [16] offers two top-level classes, whereas the taxonomies of Shen et al. [17], Liu et al. [32], and Lin et al. [31] resolve into 27, 31, and 25 fine-grained types, respectively. While this is detailed enough for benchmark diagnostics, it is too granular to be handed to a student as an actionable checklist.

Unit of analysis. The fine-grained taxonomies label a finished SQL query against a benchmark's gold query [32] or gold result [17,31]; their unit is the triple (natural-language question, database, SQL query). What an assistant returns to its user, however, is an answer consisting of a query together with a prose explanation, possibly a number presented as its result, and occasionally an assurance about its properties. Failure modes that reside in the answer rather than in the SQL text cannot be represented in the triple. These include requests with more than one valid translation resolved silently, without telling the user that a choice was made; explanations that describes a query different from the one that was written; numerical results reported even though no query was executed; and unwarranted assurance about the security of query templates. Shen et al. [17] filed ambiguous questions under "Not an Error", while Liu et al. [32] removed them during curation; none of the taxonomies reviewed in this study treat these failure modes as failures of the model.

Division of mitigation labor. No prior taxonomy ties its categories to who is expected to prevent or detect each failure. This is important in a teaching setting, where the aim is not

only to obtain a correct query but to make both the source of each failure and its remedy legible to a novice and leave students with habits that they will need when instructor-prepared support is unavailable.

1.3. Nature of the Proposal and Contributions

The proposal is a teaching instrument, not a benchmark; its contribution is conceptual and illustrative. The fifteen categories are the outcome of an expert-driven process rather than a systematic empirical derivation. They were identified by reading the text-to-SQL and LLM-hallucination literature [16,17,32–34] and by matching the failure modes documented there with errors that students make year after year in our course [20–22]. Each category is illustrated on a single common five-table PostgreSQL schema. The baseline output of each worked example is constructed by the author; it shows what an assistant could return if the failure occurred, and is not a transcript of a specific model's response (Section 2.5). This choice of method determines what the paper claims. Each worked example shows that a failure mode is plausible, as documented in prior work or in student practice, and that a countermeasure exists.

The paper does not estimate how often a failure occurs, how its frequency varies across models, or how effective a countermeasure is across requests or student cohorts. The taxonomy is not derived statistically from a corpus of failed queries, as in [17,31,32]. These quantitative questions are deliberately left to future work, since answering them presupposes an instrument of the kind proposed here. The categories are instead cross-checked against the empirically derived taxonomies of prior work, and the proposal should be judged as a teaching instrument grounded in and consistent with that work.

The mitigation strategy has two complementary layers. The first is static: a *context pack*, prepared once by the instructor, made of a set of standing rules of behavior and seven reference files describing the course database (schema DDL, column catalog, foreign key graph, value catalog, business glossary, exemplars, and reference queries). Each user stores it once in a Claude Project of their own [35], so that it is present in every conversation without being re-entered.

The second is dynamic: an *execute–observe–correct loop* in which the assistant, connected to a live PostgreSQL database through the Model Context Protocol (MCP), executes each candidate query and the checks that the standing rules prescribe for it, inspects the results, and revises the query before presenting it. The person interacting with the assistant varies (the instructor during lectures, the student in the laboratory or at home), and we call either of them the user.

Within this scope, the paper makes four contributions:

- A taxonomy of fifteen text-to-SQL hallucination categories at an intermediate granularity, fine enough to separate structurally distinct failure modes (e.g., a fabricated join path from silent row multiplication through fan-out) yet coarse enough to serve as a classroom checklist. Each category is illustrated by a worked example in two configurations: an illustrative baseline, showing what an assistant could return when even the schema DDL is omitted, and a mitigated configuration run with Claude Sonnet 5, the context pack, and the loop.
- Four categories that have no counterpart in prior text-to-SQL taxonomies because they concern the answer returned to the user rather than the SQL query: silent resolution of ambiguity, explanation–query inconsistency, fabricated execution results, and false security assurances. Two further categories, namely, row multiplication through fan-out and incomplete traversal of a hierarchy, have no dedicated counterparts, although their effects are observable as wrong results.

- A two-layer mitigation with explicit definitions of generation, loop, and session. This specifies what the assistant does between the user's question and the answer, as well as why every loop terminates.
- An assignment of each countermeasure to one or more of three actors: the instructor, who prepares the context pack; the loop, which runs the prescribed checks without user input; and for four categories, the user, who must decide what the assistant cannot decide alone (the meaning of an undefined business term, the intended reading of an ambiguous request, whether an explanation matches the executed query, and whether every reported number comes with the query that produced it). To our knowledge, none of the existing taxonomies are built for teaching SQL or divide mitigation in this way.

The mitigation is instantiated with Claude Sonnet 5, a model released after most of the studies discussed above. To our knowledge, it has not previously been examined in the text-to-SQL hallucination literature, although earlier Claude models were among those evaluated in [32,36]. This choice grounds the mitigated configuration in the affordances of a current model, namely, adaptive reasoning and native tool use through MCP. The categories themselves are expected to be properties of schema-grounded text-to-SQL generation rather than artifacts of this model, while their frequency and severity are expected to vary with the model and the quality of the context pack.

The choice of a Sonnet-tier model is also deliberate. At the time of writing, Claude Sonnet was the model made available by Anthropic on its free plan [37], whereas the higher Opus tier requires a paid subscription (about EUR 21 per month in Italy, VAT included, as of October 2026). A teaching method that presupposes a paid subscription would make the quality of each student's learning experience depend on their ability to pay, which is at odds with the values of diversity and inclusiveness and with the principle of fairness and non-discrimination set out in UNESCO's Recommendation on the Ethics of Artificial Intelligence [38]. Therefore, we adopt a zero-cost requirement: every component of the workflow proposed here must be available to every enrolled student free of charge. The requirement also bounds the scope of the proposal to classroom-size databases, for which the context pack can be read in full on the free plan.

The rest of the paper is organized as follows: Section 2 presents the materials and methods, consisting of the running PostgreSQL database (Section 2.1), the Claude Project that holds the context pack (Section 2.2), the execute–observe–correct loop (Section 2.3), the terminology used throughout (Section 2.4), and the methodology by which the taxonomy was developed (Section 2.5); Section 3 presents the fifteen categories, each with a request, the baseline output, a diagnosis, and the mitigation, then closes with the rationale for merging and splitting candidate categories; Section 4 positions the taxonomy against related error classifications, first at the level of whole works and then category-by-category; Section 5 discusses the pedagogical positioning of the proposal, the mitigations read (by actor), the model-agnostic status of the categories, the consequences of the zero-cost requirement, the scope of the proposal, and its limitations; Section 6 concludes the paper and outlines future work; finally, Appendix A gives the standing rules and the content of the knowledge files as entered in the Claude Project.

2. Materials and Methods

This section presents the materials and methods on which the taxonomy rests. Section 2.1 introduces the running PostgreSQL database against which every example is written. Section 2.2 describes the text-to-SQL workflow with Claude Sonnet 5 and the Claude Project that holds the context pack (the static layer of the mitigation). Section 2.3 describes the dynamic layer, which is the execute–observe–correct loop through which

Claude checks a query against the live database. Section 2.4 fixes the meaning of the terms used throughout the paper. Finally, Section 2.5 describes how the fifteen categories were identified, retained, merged, and distinguished from prior work.

2.1. The Running Database

All examples in this article refer to a simplified relational database schema which models a small-to-medium "order management system" typical of a retail or B2B sales operation. It is organized around five interrelated tables (Figure 1):

- `employees` stores the staff who handle sales, including a self-referencing `manager_id` that captures the internal reporting hierarchy (who supervises whom).
- `customers` holds contact details, country, and sign-up date about clients.
- `products` catalogs the items available for sale, including the category, current unit price, and stock level, thereby supporting inventory tracking.
- `orders` represents individual sales transactions, linking a customer, the employee who processed the sale, the order date, and its current status (pending, shipped, delivered, or canceled).
- `order_items` is the associative table that breaks each order into its constituent product lines, recording the quantity and unit price (which may differ from the product's current listed price) at the time of sale.

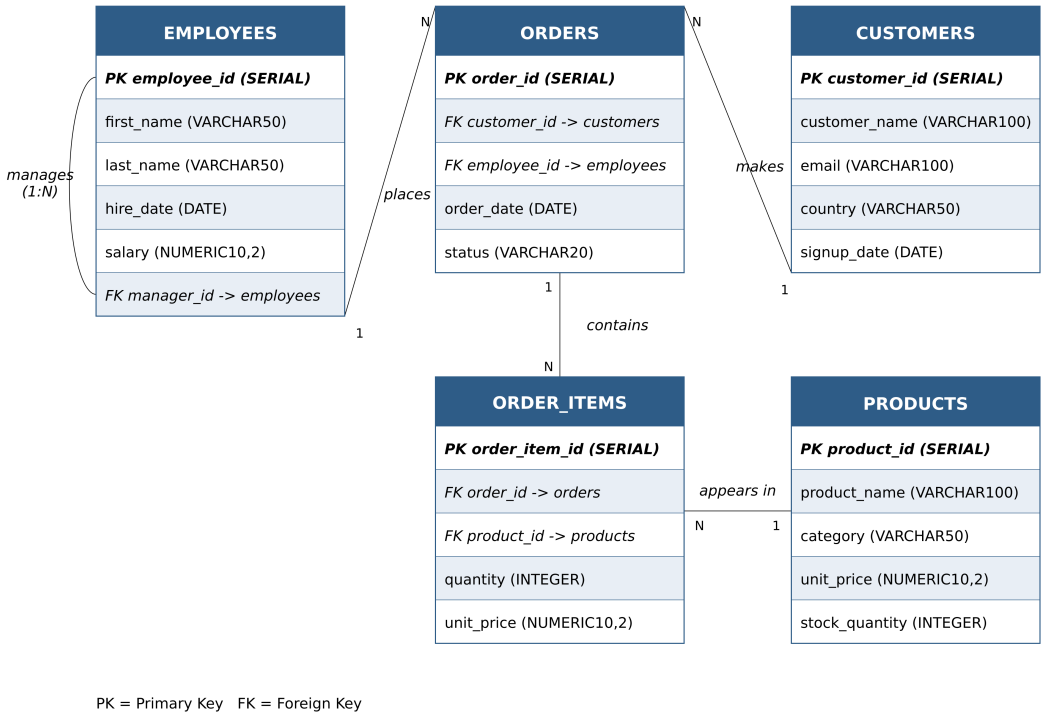

Figure 1. The E-R diagram that conceptualizes the reference relational DB.

The chosen database combines the structural features (self-referencing hierarchy, one-to-many order lines, low-cardinality status codes, mixed date, and numeric types) needed to instantiate every hallucination category with a realistic example. The schema is deliberately compact; it has fewer tables than the textbook schemas on which SQL is commonly taught [39], but is missing none of the structural features that the categories require. The PostgreSQL DDL statements defining the five tables are presented below:

```
CREATE TABLE employees (
employee_id SERIAL PRIMARY KEY,
```

```
first_name VARCHAR(50),
last_name VARCHAR(50),
hire_date DATE,
salary NUMERIC (10,2),
manager_id INTEGER REFERENCES employees(employee_id));

CREATE TABLE customers (
customer_id SERIAL PRIMARY KEY,
customer_name VARCHAR(100),
email VARCHAR(100),
country VARCHAR(50),
signup_date DATE);

CREATE TABLE products (
product_id SERIAL PRIMARY KEY,
product_name VARCHAR(100),
category VARCHAR(50),
unit_price NUMERIC(10,2),
stock_quantity INTEGER);

CREATE TABLE orders (
order_id SERIAL PRIMARY KEY,
customer_id INTEGER REFERENCES customers(customer_id),
employee_id INTEGER REFERENCES employees(employee_id),
order_date DATE,
status VARCHAR(20) -- 'pending'|'shipped'|'delivered'|'cancelled' );

CREATE TABLE order_items (
order_item_id SERIAL PRIMARY KEY,
order_id INTEGER REFERENCES orders(order_id),
product_id INTEGER REFERENCES products(product_id),
quantity INTEGER,
unit_price NUMERIC(10,2)).
```

2.2. The Claude Sonnet 5 Text-to-SQL Workflow

Claude Sonnet 5 is a general-purpose Large Language Model (LLM) developed by Anthropic and released on 30 June 2026. It is part of the Claude family of models designed for tasks spanning natural language understanding, code generation, and multi-step reasoning. It is positioned as a mid-tier model in Anthropic's current lineup, balancing computational efficiency with strong performance on complex tasks such as data analysis and problem solving. Claude Sonnet 5 supports tool use, extended reasoning, and integration into agentic workflows, making it applicable to educational contexts, including tasks that require interpreting, generating, or explaining structured queries such as SQL statements over relational databases. All the components used in this paper satisfy the zero-cost requirement stated in Section 1: the Sonnet model on the free plan, Claude Projects, the Claude Desktop application, the open-source MCP server for PostgreSQL, and PostgreSQL itself.

A Claude Project is a workspace that groups conversations around a common purpose. It has two components (Figure 2), which play different roles. The *Project instructions* consist of a short text that tells Claude how to behave in every conversation of the

Project. They contain four general rules, which apply to every query: a loop-control rule (a query is presented as verified only if it passed every prescribed check, and is revised at most three times), a schema rule (use only declared tables; execute every query, or present it as not executed), a dialect rule (PostgreSQL 17 only), and a provenance rule (report no number that an executed query did not return). They then contain one rule for each category. Instructions are *prescriptive*, meaning that they say what Claude must or must not do. The *Project knowledge* is a set of seven files uploaded to the Project. It is worth noting that this number does not grow with the size of the schema of the running database (with ten tables instead of five—a hospital instead of an order system—the same seven files are necessary; only their content changes). The seven reference files contain the schema DDL, column catalog, foreign-key graph, value catalog, business glossary, exemplars, and reference queries. In the rules, the files are referred to by their names: `01_schema.sql`, `02_column_catalog.txt`, `03_fk_graph.txt`, `04_value_catalog.txt`, `05_business_glossary.txt`, `06_exemplars.sql`, and `07_reference_queries.sql`. Knowledge is *descriptive*, meaning that it states the facts about the course database consulted by Claude when it writes or checks a query. The rules in the instructions point to these files by name. Whenever a user opens a conversation within the Project, Claude receives both components together with the user's question. The user, whether the instructor during a lecture or a student in the laboratory or at home, types only the natural-language question. Instructions do not "run" anything; in each conversation, Claude reads both components, and when a rule says, for instance, "use only the tables declared in 01_schema.sql", Claude goes to that file in the knowledge. The choice of seven files is an organizational one, not a requirement of Claude; the same information could be put in fewer files, even just one, or split into more. It is the list of seven information types that is general, with each being tied to the categories it prevents. Separate files are recommended because the rules can then cite each one by name, allowing a student to check, update, or replace one without touching the others. On the other hand, the size of the files does grow with the schema. For a classroom database (few tables), the whole knowledge is a few pages. For a very large schema, the Project's knowledge capacity and the way Claude accesses large knowledge become relevant. This is a limit of scope, not of generality, as this paper addresses classroom-size databases.

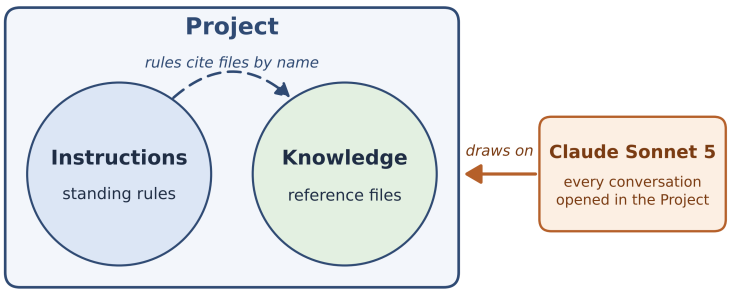

Figure 2. A Claude Project. The Project instructions hold the standing rules; the Project knowledge holds the reference files. The dashed arc indicates that the rules refer to the files by name. Claude draws on both components in every conversation opened within the Project, so neither has to be re-entered by the user.

Figure 3 shows the workflow as it plays out when Claude Sonnet 5 is used for a text-to-SQL task, organized in three stages: context preparation, model reasoning, and (optionally) agentic tool use with self-correction.

Context preparation. The model needs to know the specific database it is querying and how it is expected to behave. The instructor collects this information into a context pack made up of the Project instructions (the standing rules) and the seven reference files

of the Project knowledge described above. Each user stores it once in a Claude Project, a feature that is available on the free plan. From then on, it is included automatically in every conversation opened within that Project, and in each conversation the user supplies only the natural-language question (Figure 3). In this way, the model receives everything it needs about the database inside its context rather than through training, a setting known as in-context learning (ICL) [16,17]. This step matters most for teaching, as the quality of the schema information given to the model is normally the single biggest driver of correct output, moreso than the phrasing of the question.

Reasoning and SQL generation. Claude Sonnet 5's adaptive thinking mechanism allocates reasoning in proportion to task difficulty. For a text-to-SQL prompt, this typically involves implicit schema linking (mapping question terms to tables/columns), query planning (deciding joins, aggregation, filtering order), and drafting the SQL statement together with a natural-language rationale. This is useful in class for showing students how a query maps back to the question.

Agentic tool use and self-correction (optional). This stage is optional only in the sense that it requires a database connection; when the connection is present, the standing rules make it mandatory for every query. By connecting Claude Sonnet 5 to a database tool, it can ask the tool to run the candidate query against the database; Claude then inspects the returned rows or error message and revises the query before answering, shown as the dashed feedback loop in Figure 3. The formalization of this workflow came with the seminal paper by Yao et al. [40]. Embedding this loop directly into instruction reflects what a student is asked to do manually (running, inspecting, and correcting code); showcasing this process offers greater pedagogical value than presenting the end-result query.

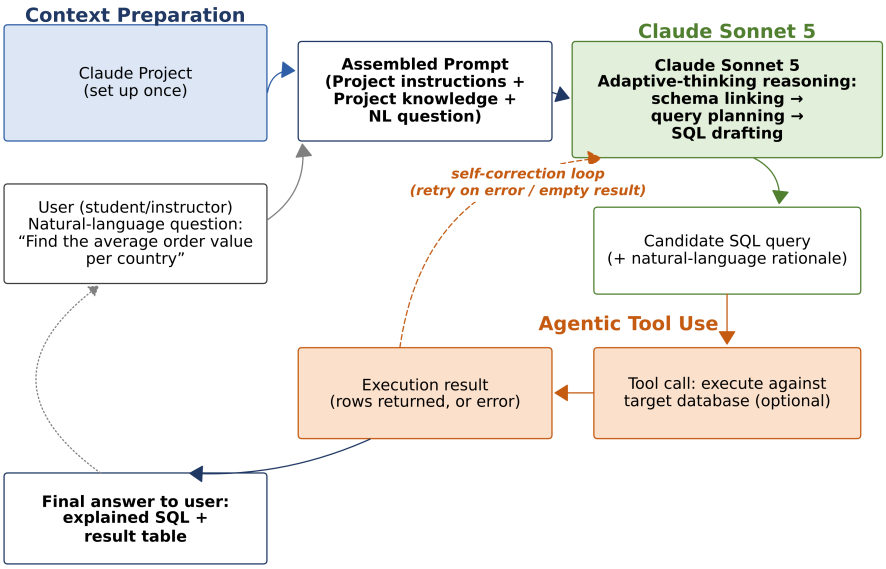

Figure 3. Claude Sonnet 5 text-to-SQL workflow, from natural-language question to verified answer.

2.3. The Agentic Execute-Observe-Correct Loop

Where Section 2.2 described the static layer (the Project), this section describes the dynamic layer (the loop). Figure 3 represents the optional agentic execution stage as a single dashed feedback arrow, without unpacking what actually happens along it. The five-step diagram of Figure 4 expands that arrow. The same five-step loop recaps what happens whenever Claude checks any assumption against the database: whether a table or column exists, whether a join has the cardinality it appears to have, whether an aggregation grain is correct, whether an enumerated value actually occurs in the data, or whether a

filter behaves as expected under NULLs. For this reason, the diagram is presented here in a category-neutral form. The five steps are as follows:

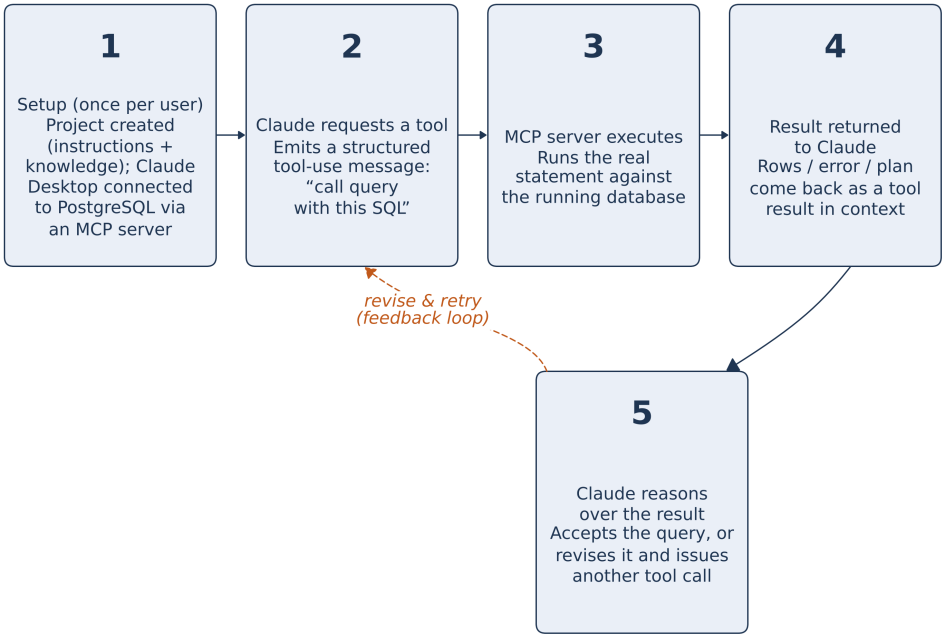

Figure 4. The generic agentic tool use loop underlying the dashed feedback arrow in Figure 3, applicable to any of the fifteen hallucination categories considered in this paper.

1. One-time setup outside the conversation. During a lecture, the instructor shows the class the content of the context pack and two one-time operations, which each student then repeats with their own account and computer: (i) creating a Claude Project, pasting the standing rules into the Project instructions and uploading the seven reference files to the Project knowledge (Section 2.2). (ii) connecting the Claude Desktop application to the course PostgreSQL instance through an MCP server for PostgreSQL. The Project belongs to the student’s Claude account; the MCP connection is configured on each computer from which the student will query the database. Neither operation involves programming; the server already knows how to run queries and return results, plans, or errors. MCP is an open standard that lets an assistant such as Claude connect to an external tool (here, a PostgreSQL database) through a small ready-made server, with no application code to write.
2. Claude requests a tool call; it does not run anything. When the standing rules require a check or when Claude decides an additional check is needed (e.g., a catalog lookup), it does not execute SQL itself; it emits a structured tool use message through MCP. That message is still just text produced by the model, formatted as a function call request.
3. The MCP server runs the real statement. The server intercepts that request, sends the actual statement to the PostgreSQL instance, and gets back the result consisting of rows, an error message, or when the statement is an `EXPLAIN`, the planner’s execution plan.
4. The result is fed back into Claude’s context. MCP returns that result to the conversation as a “tool result”. Only now does Claude “see” it, as ordinary text in its context window, just as it reads any other piece of information supplied to it.
5. Claude reasons over that text. It reads the returned rows, error, or plan and checks whether they match what it expected. If they do, and if every check that the standing rules prescribe for the query has been run and passed, then it presents the query as verified; otherwise it revises the query and issues another tool call, closing the

loop back to Step 2. A query that still fails a check after three revisions is not revised further. Claude presents the last version as not verified and names the check that failed. Thus, *Verified* means that the query passed the prescribed checks, not that it is proved correct; a check establishes only what it tests.

Without an MCP server (or an equivalent tool) connected to the running database, none of this loop is available. Claude alone can only guess at what a query would return. In general, the "execute → observe → revise cycle" can repeat more than once, but not indefinitely (Step 5).

2.4. Terminology and Core Concepts

Five terms recur throughout this paper: user, context pack, generation, loop, and session. We fix their meaning here.

User. In the remainder of this article, the term "user" denotes whoever is interacting with Claude at a given moment. In the setting considered here, both the instructor and the students play this role at different times: the instructor while demonstrating queries and failure modes during lectures, and the students while working in the laboratory or studying at home. Since each of them works within their own Claude Project holding the same context pack, the workflow described in the following sections is the same in all three situations. The terms "instructor" and "student" are used only where the distinction matters, for instance when a mitigation is assigned to an actor.

Context pack. We call the content of the Claude Project described in Section 2.2 the "context pack", denoted as P; $P = (I, K)$, where I is the Project instructions and K is the Project knowledge. The instructor prepares P once, and it is stored by each user in their own Project; unless the instructor releases a new version, it does not change during a session or between sessions. For classroom-size schemas, P is read in full during every conversation (Section 1.3).

Generation. A generation is a single response of the model. Claude reads everything currently in its context and produces one output, which can take only one of two forms: a final answer, which is presented to the user, or a request to run a statement on the database (a tool call), which is not shown to the user as an answer. In the latter case, the MCP server runs the statement and returns its result, consisting of rows, an error message, or an execution plan. The result is added to the context, and the model is invoked again. The second invocation is a new generation. Answering one question may take several generations, but each generation either answers or asks for a statement to be run. Table 1 shows an example of three generations that answer the question "Count how many orders are completed". Each generation reads the context accumulated thus far and produces one output; G_1 and G_2 end with a tool call, the result of which is added to the context of the next generation, and G_3 ends with the final answer.

Table 1. Generations answering one question.

Generation	What Claude Sees	What Claude Produces	Type
G_1	context pack + question	`SELECT DISTINCT status FROM orders`	tool call → server returns: pending, shipped, delivered, cancelled
G_2	the above + probe + its result	`SELECT COUNT(*) FROM orders WHERE status = 'delivered'`	tool call → server returns the count
G_3	the above + query + count	the query, the count, and a note that "completed" was mapped to delivered	final answer

Formally, a generation is a function

$$G_i : C_i \to (A_i, \tau_i), \tag{1}$$

where:

- C_i is the context at generation i, containing the context pack P, the question currently being answered, and the history H_i of everything that happened earlier in the conversation (previous questions and answers, tool requests and their results).
- A_i is the text the produced by the model, consisting of an SQL statement, a rationale, or an answer.
- τ_i is either `none` or `run`(s), where s is the statement to be executed; s is either the candidate query or a verification probe, such as a catalog lookup, a DISTINCT probe, a row count, or an EXPLAIN. If $\tau_i =$ `run`(s), the server returns a result R_i and the next generation sees it: $C_{i+1} = C_i \cup \{A_i, s, R_i\}$. If $\tau_i =$ `none`, then A_i is the answer shown to the user.

Two clarifications are necessary to complete the definition. First, the model's internal adaptive thinking reasoning happens inside a generation and does not create additional ones. Second, running the statement (Steps 3–4 of Figure 5) happens between two generations and is carried out by the MCP server and the database, not by the model.

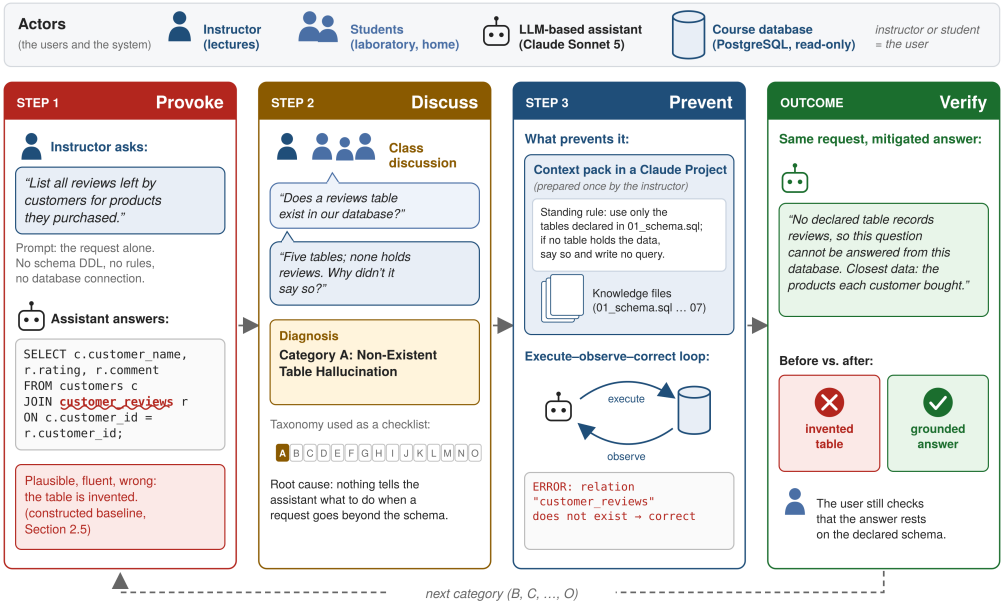

Figure 5. Classroom use of a worked example, illustrated with Category A. The baseline is constructed (Section 2.5); the mitigated answer is that of the configuration of Section 2.5.

Loop. The execute–observe–correct loop for a question is the run of consecutive generations that answer it, G_1–G_3 in Table 1. Every generation in the run except the last requests a statement; the last one answers. No user turn occurs inside a loop. In a conversation without a database connection (i.e., one opened on a computer where MCP is not configured), each question is answered by a single generation and every loop has length one. Every loop ends in one of three ways: with a verified query; with a query presented as not verified, together with the check that it failed; or, in a conversation without a database connection, with a query presented as not executed, as the general standing rules require. Because each candidate query is followed by the finite set of checks that the rules prescribe for it and since at most four candidates are written (the first and three revisions), every loop is finite. Like every standing rule, the cap is an instruction to the model (Section 2.2), and the number of revisions is visible to the user in the tool calls of the conversation.

Session. A session is one conversation opened within the Project. Its initial context is $C_0 = P$. During a session the user asks a sequence of questions $Q_1, \ldots, Q_m$, and each question is answered by one loop. A session is bounded by the conversation, not by time: reopening the same conversation later continues the session, whereas opening a new conversation starts a new one. Formally, a session is the triple

$$S = \langle P, (Q_1, \ldots, Q_m), (G_1, \ldots, G_n) \rangle, n \geq m, \quad (2)$$

where $G_1, \ldots, G_n$ are the generations of all its loops in the order in which they occur. The equality $n = m$ holds only when no question triggers a tool call, that is, when every loop has length one.

Consequences. Because the context pack resides within the Project, it is part of C_0 in every session opened within it. The user never re-supplies it, whether between the generations $G_1, \ldots, G_n$ of one execute-observe-correct loop (where no user turn occurs) or across sessions (for instance, when a student returns to an exercise days later). Placing the pack in the Project instructions rather than in an ordinary chat message also keeps it out of the part of a long conversation that may be summarized to fit the context window.

2.5. Taxonomy Development Methodology

The taxonomy proposed in this paper is a working expert-driven taxonomy. This section describes how the categories were identified, retained, merged, excluded, and distinguished from prior work.

Identification of candidate categories. The candidate failure modes were identified from two complementary sources; the worked examples were then constructed and checked as described below.

- *Literature*. We reviewed the error classifications of Qu et al. [16], Shen et al. [17], and Liu et al. [32], the hallucination survey of Huang et al. [34], and the work of Saparina and Lapata [33] on linguistic ambiguity, extracting the failure modes that can arise in schema-grounded SQL generation.
- *Teaching experience.* We drew on 27 years of experience teaching SQL in database courses to Master's students in Management Engineering at L'Aquila University (Italy), from which we took the errors that novices recurrently make, as also documented by large-scale and think-aloud studies of student SQL errors [20–22], and that an assistant may reproduce or fail to correct.
- *Baseline outputs.* The baseline output reported for each category in Section 3 is not a transcript of a model's response; it was constructed by the author, starting from the errors that students recurrently make in our course and from the failure types documented in prior work [16,17,20–22,31,32], and is written in the form that an LLM-based assistant could give to the request when not even the schema DDL is supplied in the conversation and when no standing rules, knowledge files. or database connection are available. Each baseline should be read as a conditional scenario showing what the user would receive if the failure occurred; this offers the instructor a concrete starting point for class discussion around failures that students may meet when using any of the LLM-based assistants in circulation. Whether and how often a given assistant actually produces such failures will depend on the model, its version, and the context it receives, which we do not cover here (Section 1.3).
- *Mitigated configuration.* The mitigation of each category was checked with Claude Sonnet 5 through the Claude Desktop application (free plan), with the full context pack stored in the Project and Claude Desktop connected to PostgreSQL 17.9 (Windows x64) through the reference MCP server for PostgreSQL (@modelcontextprotocol/server-

postgres version 0.6.2, MIT licence), connecting as a dedicated database role with read-only privileges. The candidate failure modes were consolidated into fifteen categories through the criteria below.

Note on the MCP server. By the time of our sessions, this server had been archived by its maintainers and marked as deprecated on npm, and it no longer receives updates; we report this for transparency. It does not affect the taxonomy, as neither the categories nor their baseline outputs depend on any MCP server. In the mitigated configuration, the server's only function is to execute the statements requested by Claude and return their results. Its read-only safeguard is implemented by the server itself, which wraps each statement in a read-only transaction. We did not rely on that safeguard alone; the server connected through a role to which PostgreSQL itself grants only read privileges, on a local instance holding only course data. For future classroom deployments, we recommend a maintained MCP server for PostgreSQL, combined with the same kind of read-only role.

Inclusion criteria. A candidate was retained only if it satisfied all of the following conditions, which we applied through expert judgment rather than a formal procedure: (a) it can be instantiated on the running PostgreSQL schema with a concrete natural-language request and a baseline output for which the stated engine behavior (error or silent result) holds on that schema; (b) it is structurally distinct from the other categories, i.e., it requires a different diagnosis or a different countermeasure; (c) it is actionable at classroom level, i.e., it can be paired with at least one of a standing rule in the Project instructions, an entry in one of the seven knowledge files, or a check in the execute–observe–correct loop.

Merging and splitting. Candidates sharing the same root cause and the same countermeasure were merged. Candidates sharing a symptom or locus but differing in mechanism were kept separate.

Exclusions. The taxonomy is deliberately limited to failure modes that are grounded in the schema or in the interaction between user, model, and database and that are mitigable at classroom level. Therefore, we did not create separate categories for the following families of errors:

- Purely syntactic slips (unbalanced parentheses, misplaced keywords, missing quotes). The parser reports them immediately, they do not depend on schema grounding, and they are covered in [17] (Syntax Error). The execute–observe–correct loop handles them without dedicated context.
- Surface-level result-shaping errors (projection, ORDER BY direction, LIMIT, DISTINCT). They are extensively catalogued in [17,32] and are not distinctive in a teaching setting, since the remedy is to restate the request rather than to add context or verification.
- Generic logic slips that do not depend on information missing from the prompt (e.g., a wrong comparison or set operator, = used against a multi-row subquery). In PostgreSQL several of them raise engine errors and are caught by the same loop.
- Performance problems (missing indexes, inefficient plans), because they yield correct results.
- Benchmark artifacts (erroneous gold queries, foreign-key violations in the data). They are irrelevant here because our examples are not scored against a gold query.
- Constructs beyond the scope of the running schema (window functions, materialized views, multi-statement transactions, data modification and DDL statements).

3. Results: Taxonomy of Hallucination Categories

This section presents the fifteen categories, labeled A to O. Each is described with the same elements. The request is the natural-language question given to Claude. The baseline output is the illustrative output constructed as described in Section 2.5, consisting of what

an assistant could return if the failure occurred. It is a prompt for classroom discussion, not a record of a specific model's response; accordingly, where the issue states what 'the model' does, it describes this scenario, not an observed run. The issue then explains what is wrong with that output. The mitigation states what the mitigated configuration of Section 2.5 adds: the standing rule entered in the Project instructions, quoted verbatim; the files of the Project knowledge to which the rule points (Section 2.2); and the check that Claude carries out at run time within the execute–observe–correct loop (Section 2.3). Where part of the mitigation is left to the user (Section 2.4), it is stated last. Although the baseline omits even the schema DDL (Section 2.5), no category below is caused merely by its absence. Each arises from something that the DDL, even when supplied, does not provide: a rule of behavior, a fact about the data or the business that no CREATE TABLE statement records, or the evidence that only execution can give.

Figure 5 shows how each worked example is used in class: the instructor puts the request to an assistant without context and shows the baseline (Step 1); the class diagnoses the failure with the taxonomy as a checklist (Step 2); the instructor shows what the context pack and the loop add (Step 3); and the same request, now mitigated, returns an answer grounded in the schema, which the user still checks (Outcome).

Table 2 gives an overview of the fifteen categories before they are described one-by-one. For each category, the table states the symptom, whether executing the query of the baseline reveals the error, and the actors involved in the mitigation. Only three categories (A, B, and G) produce an error that the engine reports. In the other twelve, the query executes and returns a result that looks plausible, while in Category N no query is executed at all. For these twelve, execution alone is of no help; the loop exposes the error only through the checks that the standing rules prescribe, which is why each category below pairs a rule with a run-time check. In four categories (F, K, M, and N), part of the mitigation is left to the user.

Table 2. Overview of the fifteen hallucination categories. The fourth column states whether the engine reports an error when the query of the baseline is executed, and if not, what the execution returns. I = instructor (prepares the context pack once); L = execute–observe–correct loop (no user input); U = user, i.e., whoever is interacting with Claude at run time (the instructor during lectures, the student in the laboratory or at home).

Cat.	Name	Symptom	Does Execution Reveal the Error?	Actor(s)
A	Non-Existent Table.	Invented table absent from the schema.	Yes: relation does not exist.	I, L
B	Non-Existent Column.	Invented or derived column treated as stored.	Yes: column does not exist.	I, L
C	Fabricated Join-Path.	Join along a relationship that the schema does not declare.	No: plausible row count.	I, L
D	Cardinality/Fan-Out.	Two one-to-many branches joined before aggregating: rows multiplied.	No: plausible totals.	I, L
E	Aggregation-Scope.	Aggregate computed one grain too low.	No: well-typed number.	I, L
F	Enumerated-Value.	Filter on a category value absent from the data.	No: returns 0.	I, L, U
G	Dialect/Function.	Function or syntax of another DBMS.	Yes: function does not exist.	I, L
H	Data-Type.	Integer division silently truncates a ratio.	No: returns 0.	I, L

Table 2. *Cont.*

Cat.	Name	Symptom	Does Execution Reveal the Error?	Actor(s)
I	NULL-Semantics.	NULL tested with = or <>.	No: empty result.	I, L
J	Temporal-Logic.	Period boundary computed by hand: a day lost or added.	No: plausible rows.	I, L
K	Ambiguity-Induced.	Ambiguous request resolved silently in one way.	No: every reading runs.	I, L, U
L	Hierarchical/Recursive-Structure.	Single self-join for a request over all levels of a hierarchy.	No: a subset of the answer.	I, L
M	Explanation–Query Inconsistency.	Explanation describes a query different from the one written.	No: empty result.	I, L, U
N	Fabricated-Execution/Overconfident-Result.	Result reported although no query was executed.	No: nothing is executed.	I, L, U
O	Security-Assumption.	Injectable query template declared safe.	No: runs with ordinary input.	I, L

A. Non-Existent Table Hallucination

Request:

"List all reviews left by customers for products they purchased."

Baseline output:

```
SELECT c.customer_name, r.rating, r.comment
FROM customers c JOIN customer_reviews r ON
     c.customer_id = r.customer_id;
```

Issue: The schema declares five tables, none of which records reviews. The request presupposes such a table nonetheless. Instead of reporting that the database cannot answer the question, the model invents one, `customer_reviews`, with columns (rating, comment) typical of review tables seen during training. Supplying the DDL would not by itself prevent this: it describes what exists, but says nothing about what to do when a request goes beyond it, and the most fluent continuation is a plausible query. The query is syntactically well-formed and reads as sound domain logic.

Mitigation.

Standing rule (Project instructions):

> Use only the tables declared in `01_schema.sql`. If a request needs data that no declared table holds, say so and do not write a query for it. Before presenting any query, execute it with the PostgreSQL MCP tool; if the engine reports that a relation does not exist, revise the query and execute it again. If no database tool is available in this conversation, present the query as not executed.

This rule is one of the general standing rules of Section 2.2: it applies to every query, not only to requests of Category A.

Knowledge: `01_schema.sql`, the file the rule names. Its content is the schema DDL that the baseline lacks. What changes is not only that the DDL is now available, but that a rule binds Claude to it and that, being part of the Project, it belongs to the initial context of every session (Section 2.4).

Run-time check: Claude executes the candidate query through MCP and reads the error returned by PostgreSQL: relation `customer_reviews` does not exist. The rule then

determines the outcome. Rather than replacing the invented table with another guess, Claude answers that no declared table records reviews, and may point to the closest data the schema does hold, namely, which products each customer bought. Execution alone would only reveal that the query fails; the rule is what makes "this question cannot be answered from this database" an admissible final answer. With the rule in place, Claude normally declines before drafting any query; the execution described here is the backstop for the case in which it drafts one nonetheless.

B. Non-Existent Column Hallucination

Request:

"Show the total amount of each order."

Baseline output:

```
SELECT order_id, total_amount
FROM orders;
```

Issue: The schema lists the columns of `orders`, and `total_amount` is not among them; the model writes it nonetheless, because the column is common in order schemas seen during training.

Mitigation:

As in Category A, the mitigation combines a standing rule, bound to a knowledge file, with a check that Claude runs during the loop.

Standing rule (Project instructions):

> Reference only columns listed as STORED for that table in `02_column_catalog.txt`. If a requested value is not STORED (e.g., a total, a revenue, a margin), compute it with the derivation given in the catalog and say so in a comment. Before presenting, verify every (table, column) pair against `information_schema.columns`.

Without the column catalog and this rule, the model falls back on patterns from generic schemas seen during training, where an `orders.total_amount` column is extremely common.

Knowledge: The DDL alone is not enough for columns. A `CREATE TABLE` statement tells the model which columns exist but not what they mean, and it is this gap that lets the model treat `total_amount` as a stored column. The column catalog, one of the seven knowledge files of Section 2.2 (`02_column_catalog.txt`), closes the gap. For every attribute of the five tables, it records (a) whether the attribute is STORED or must be DERIVED, together with the derivation of each derived attribute, e.g., `order value = SUM(quantity × unit_price)` over `order_items`, per `order_id`, along with (b) a one-line business meaning, where the name does not make it obvious.

Run-time check: The agentic loop of Figure 5 adds what the rule alone cannot guarantee. Before presenting a query, Claude checks every (table, column) pair it references against `information_schema.columns`. Executing the query would also expose this error, since PostgreSQL reports column `total_amount` does not exist. However, the catalog check also confirms the remaining pairs and identifies the offending one before any result is shown. While the knowledge file and the rule only reduce the probability of the hallucination, this check detects it mechanically.

The query that the rule produces follows

```
-- order value is DERIVED (02_column_catalog.txt):
-- SUM(quantity * unit_price) per order_id
SELECT o.order_id,
    COALESCE(SUM(oi.quantity * oi.unit_price), 0) AS order_value
FROM orders o
```

```
    LEFT JOIN order_items oi ON oi.order_id = o.order_id
GROUP BY o.order_id;
```

This mitigation is pedagogically meaningful, since students learn to read the DDL as the single source of structural truth and the column catalog as a separate semantic layer, which is closer to how they will work with real schemas later.

C. Fabricated Join-Path Hallucination

Request:

"For each customer, list the products they bought."

Baseline output:

```
SELECT c.customer_name, p.product_name
FROM customers c JOIN products p ON c.customer_id = p.product_id;
```

Issue: The model invents a direct relationship between `customers` and `products` that was never declared, while the true path runs through two intermediate tables, `orders` and `order_items`, along three foreign-key edges. The schema DDL declares every one of these edges in its `REFERENCES` clauses; the model nonetheless joins two integer columns that merely look related. Because `customer_id` and `product_id` are both `SERIAL` columns starting at 1, the join returns a real and non-empty plausible-looking row count, with nothing about the count being anomalous on its face.

Mitigation:

Standing rule (Project instructions):

> Join two tables only along the edges listed in `03_fk_graph.txt`, directly or through a chain of listed edges; when no direct edge exists, use the mandatory path given there. A shared data type or a similar column name is not evidence of a relationship. Before executing, check the column pair of every join predicate against the foreign keys declared in `pg_catalog.pg_constraint`, and revise any join whose pair is not a declared edge.

Knowledge: `03_fk_graph.txt` lists the five direct edges and the mandatory path for every pair of tables without a direct edge, and also states that no other relationship exists. It restates in graph form what the DDL declares edge-by-edge while adding what the DDL leaves implicit, namely, which pairs of tables are unrelated and which path is mandatory when a relationship exists only through intermediates.

Run-time check: A fabricated join path cannot be recognized from its result, as it executes without error and returns a plausible number of rows; however, it can be recognized from its structure. Before executing the query, Claude reads the foreign keys declared in the system catalog and checks that the column pair of every join predicate is one of them. The pair (`customers.customer_id`, `products.product_id`) is not, so the join is revised along the mandatory path before any result is shown. The check is complete for the running schema, and all of the schema's legitimate joins follow declared foreign keys; a schema in which some meaningful joins use non-key attributes would need those joins listed in `03_fk_graph.txt` as well.

D. Cardinality/Fan-Out Hallucination

Fan-out arises when a parent table has two independent one-to-many relationships and a query joins both of them before aggregating. In the running schema, this occurs at employees, the origin of two independent "many" branches: `orders.employee_id` → `employees.employee_id` (an employee processes many orders) and `employees.manager_id` → `employees.employee_id`, a self-reference (a manager has many direct reports). A query that computes one metric on each branch by joining both in the same `FROM` clause combines every row of one branch with every row of the other (i.e., the Cartesian product).

Request:

"For each employee, show the total revenue of the orders they handled and the number of their direct reports."

Baseline output:

```
SELECT e.employee_id, e.first_name, e.last_name,
     SUM(oi.quantity * oi.unit_price) AS revenue,
     COUNT(r.employee_id) AS n_direct_reports
FROM employees e
     JOIN orders o ON o.employee_id = e.employee_id
     JOIN order_items oi ON oi.order_id = o.order_id
     LEFT JOIN employees r ON r.manager_id = e.employee_id
GROUP BY e.employee_id, e.first_name, e.last_name;
```

Issue: The query is syntactically valid and executes without error. Withoin the same `FROM` clause, it joins two independent one-to-many chains that share the same parent (`e.employee_id`). Since nothing relates an employee's order lines to his or her direct reports, the join pairs every order line with every direct report: an employee with three order lines and two direct reports yields $3 \times 2 = 6$ rows. For an employee with no direct reports, both values are correct. For an employee with $r \geq 1$ direct reports and l order lines, the join yields $r \times l$ rows before grouping: revenue is multiplied by r, and n_direct_reports by l. Revenue is wrong whenever $r \geq 2$, and the number of direct reports is wrong whenever $l \geq 2$. No error is raised, and none of the values appear to be anomalous. The schema DDL declares both relationships correctly; the failure lies in combining them, which no declaration can prevent. In addition, the inner joins drop the employees who handled no orders, even though the request asks for each employee.

Mitigation:

Standing rule (Project instructions):

> Grain safety. When a query combines two or more one-to-many relationships that start from the same parent table, aggregate each relationship to the parent's grain in its own CTE before joining the results. State the grain of the final result in a comment (e.g., "one row per `employee_id`"). Before presenting, verify that the combined result (without an outer `GROUP BY`) has `COUNT(*) = COUNT(DISTINCT parent key)`, and that each metric equals the same metric computed from its own branch alone.

Knowledge: `03_fk_graph.txt` lists the one-to-many relationships for each parent table: `employees` → `orders`; `employees` → `employees` (as manager); `orders` → `order_items`; `customers` → `orders`; `products` → `order_items`. Thus, a parent with two "many" branches (here, `employees`) is visible at a glance. The DDL contains the same relationships, but scattered over the `REFERENCES` clauses of the referencing tables one table at a time.

Run-time check: Following the rule, Claude writes one CTE per branch, each at the grain of one row per `employee_id`, and joins the two collapsed results to `employees`:

```
-- one row per employee_id
WITH revenue_per_employee AS (
      SELECT o.employee_id,
             SUM(oi.quantity * oi.unit_price) AS revenue
      FROM orders o JOIN order_items oi ON oi.order_id = o.order_id
      GROUP BY o.employee_id),
     reports_per_employee AS (
```

```
        SELECT manager_id AS employee_id,
               COUNT(*) AS n_direct_reports
        FROM employees
        WHERE manager_id IS NOT NULL
        GROUP BY manager_id)

SELECT e.employee_id, e.first_name, e.last_name,
       COALESCE(rev.revenue, 0) AS revenue,
       COALESCE(rep.n_direct_reports, 0) AS n_direct_reports
FROM employees e
       LEFT JOIN revenue_per_employee rev ON
            rev.employee_id = e.employee_id
       LEFT JOIN reports_per_employee rep ON
            rep.employee_id = e.employee_id;
```

The left joins from `employees` also keep the employees who handled no orders. Claude then runs the two checks of the rule as additional tool calls. The first counts the rows of the combined result: since the query has no outer `GROUP BY`, one row per employee shows that neither branch has multiplied the other. The second recomputes, for every employee, the revenue from `orders` and `order_items` alone and the number of direct reports from `employees` alone, and compares both with the values returned. Both checks are needed in this form. Applied to the result of the baseline query, a row count would detect nothing; its `GROUP BY` returns one row per employee whether or not the rows were multiplied before grouping. The comparison with each branch would instead fail for every employee whose values were multiplied, i.e., every employee with at least two direct reports or with at least one direct report and at least two order lines.

E. Aggregation-Scope Hallucination

Request:

"Find the average order value per country."

Baseline output:

```
SELECT c.country, AVG(oi.quantity * oi.unit_price) AS avg_value
FROM customers c
     JOIN orders o ON o.customer_id = c.customer_id
     JOIN order_items oi ON oi.order_id = o.order_id
GROUP BY c.country;
```

Issue: The requested metric, average order value, is defined at the order grain: one value per `order_id`, obtained by summing the lines of that order. The query instead averages `quantity * unit_price` over the rows of `order_items`, that is, at the line grain: every line contributes separately, so orders with more lines weigh more. Suppose that one order of a given country totals 100 over two lines of 30 and 70 and a second order of the same country totals 50 in a single line. The query returns (30 + 70 + 50) / 3 = 50, an average line value; the request asks for (100 + 50) / 2 = 75, the average order value. The two diverge whenever orders have different numbers of lines, which is the normal case. PostgreSQL raises no error; the statement executes cleanly and returns a well-typed number. The schema DDL declares `quantity` and `unit_price`, from which order value is computed, but not the grain at which the metric is defined, which is a business definition, not a structural fact. This category is distinct from Category D: there, two one-to-many branches are joined before aggregating, and rows are multiplied, while here a single one-to-many relationship (`orders` to `order_items`) is joined, no row is duplicated, and yet the aggregation is computed one level too low. The model takes "aggregate per country" as satisfying "average order value per country", without recognizing that order value is itself the result of a prior aggregation.

Mitigation:

Standing rule (Project instructions):

> Aggregation scope. When a request asks for an aggregate per X of a metric that is itself the result of an aggregation (e.g., average ORDER value per country, where order value is the sum of an order's lines), first compute the metric at its own grain in a CTE, as defined in `02_column_catalog.txt` and `05_business_glossary.txt`; state that grain in a comment; and only then aggregate per X. Before presenting, verify that, in every group, the number of values entering the outer aggregate equals the number of distinct keys at the metric's grain (e.g., `COUNT(DISTINCT order_id)` per country).

Knowledge: `02_column_catalog.txt`, already used for Category B, records the derivation of order value and the grain of each metric, order value at the order grain and line value at the order-line grain. `05_business_glossary.txt` defines "average order value" as the mean, over orders, of the order value.

Run-time check: Following the rule, Claude writes the query in two stages. The first, a CTE, collapses order_items to one row per order; the second averages the order values per country:

```
WITH order_totals AS (
-- one row per order_id
     SELECT o.order_id, o.customer_id,
          SUM(oi.quantity * oi.unit_price) AS order_value
     FROM orders o JOIN order_items oi ON oi.order_id = o.order_id
     GROUP BY o.order_id, o.customer_id )

SELECT c.country, AVG(ot.order_value) AS avg_order_value
FROM order_totals ot
     JOIN customers c ON c.customer_id = ot.customer_id
GROUP BY c.country;
```

With an additional tool call, Claude then counts, for every country, the values that enter AVG and the distinct orders. In the two-stage query, the counts coincide because `order_totals` has one row per order. In the baseline query, they differ wherever an order has more than one line: in the illustration above, three values enter the average of a country that has two orders. This check exposes a failure that execution alone does not report.

F. Enumerated-Value Hallucination

Request:

"Count how many orders are completed."

Baseline output:

```
SELECT COUNT(*) FROM orders WHERE status = 'completed';
```

Issue: The model assumes a value for the status column ('completed') that reads as natural English but does not match the actual domain of stored values in this schema ('pending', 'shipped', 'delivered', 'canceled') (Section 2.1). PostgreSQL raises no error; the query returns 0, which reads as a legitimate answer ("no order has been completed yet").

Mitigation:

Standing rule (Project instructions):

> Before filtering an enumerated column, run `SELECT DISTINCT` on it to confirm its current values (`04_value_catalog.txt` may be out of date). If the user's word differs from a stored value only in spelling or case (e.g., 'canceled' vs 'cancelled'), use the stored value and say so. If it differs in meaning (e.g., 'completed'), apply the definition in `05_business_glossary.txt`; if the term is not defined there, ask the user. Never filter on a value that the probe did not return.

Knowledge: `04_value_catalog.txt` (the stored values of every low-cardinality column, with the date of the snapshot) and `05_business_glossary.txt`, which defines a completed order as an order with status 'delivered'.

Run-time check: The three generations of Table 1 (Section 2.4) are this check at work: G_1 probes the column, G_2 filters on the value that the glossary assigns to "completed", and G_3 reports the result together with the mapping.

G. Dialect/Function Hallucination

Request:

"Show each customer's name and signup month as month-year, and show an empty string where the email is missing"

Baseline output:

```
SELECT customer_name,
    DATE_FORMAT(signup_date, '%m-%Y') AS signup_month,
    IFNULL(email, '') AS email
FROM customers;
```

Issue: `DATE_FORMAT` and `IFNULL` are MySQL functions; PostgreSQL has no such built-ins and uses `TO_CHAR` and `COALESCE` instead. The schema DDL is PostgreSQL DDL (`SERIAL`, `NUMERIC`), but nothing in it, even when supplied, states the target dialect, and the model blends syntax across the dialects it has seen during training.

Mitigation:

Standing rule (Project instructions):

> Target dialect: PostgreSQL 17 only. Do not use functions or syntax of other DBMSs. If the engine reports that a function does not exist, rewrite the query with the PostgreSQL equivalent and execute it again.

Knowledge: None; the rule is self-contained, as the target dialect is a rule of behavior rather than a fact about the database.

Run-time check: PostgreSQL reports that function `date_format(date, unknown)` does not exist. Claude rewrites the expression and executes again; if it corrects only the reported function, a second cycle exposes `IFNULL`, in line with Section 2.3, where the cycle may repeat more than once. The final query is:

```
SELECT customer_name,
    TO_CHAR(signup_date, 'MM-YYYY') AS signup_month,
    COALESCE(email, '') AS email
FROM customers;
```

H. Data-Type Hallucination

Request:

"What fraction of orders has been cancelled?"

Baseline output:

```
SELECT COUNT(*) FILTER (WHERE status = 'cancelled')/COUNT(*)
      AS cancelled_share
FROM orders;
```

Issue: Both counts are integers, and PostgreSQL divides integers by truncation: the result is 0 whenever fewer than all orders are canceled. The query executes without error and returns a well-typed number. The DDL declares every column type, but the type of an expression such as `COUNT(*)` and the semantics of integer division are not in the DDL.

Mitigation:

Standing rule (Project instructions):

> Before dividing, make sure at least one operand is `NUMERIC`: division between integers truncates. Flag for the user any predicate that compares columns of different declared types.

Knowledge: none. The rule is self-contained: the semantics of integer division is a property of the engine, not of the course database.

Run-time check: Execution does not expose this error, since the baseline returns 0, a well-typed number. Therefore, Claude checks the type of the expression rather than its value:

```
SELECT pg_typeof(COUNT(*) FILTER (WHERE status = 'cancelled')/COUNT(*))
FROM orders; -- returns bigint:  integer division
```

It then casts one operand to `NUMERIC` and executes again. A plausibility check closes the loop; the share must be greater than 0 whenever the numerator, executed alone, is greater than 0. The final query is:

```
SELECT COUNT(*) FILTER (WHERE status = 'cancelled')::numeric/COUNT(*)
       AS cancelled_share
FROM orders;
```

I. NULL-Semantics Hallucination

Request:

"List employees who do not have a manager assigned."

Baseline output:

```
SELECT * FROM employees WHERE manager_id = NULL;
```

Issue: `manager_id = NULL` never evaluates to true in three-valued SQL logic; the query silently returns zero rows instead of raising an error, which is far more dangerous than a syntax failure because it looks like a valid and executed empty result.

Mitigation:

Standing rule (Project instructions):

> Test `NULL` only with `IS NULL`/`IS NOT NULL`, never with = or <>, following the pattern in `06_exemplars.sql`. For any filter on a nullable column, compare the row count of your predicate with the row count of its `IS NULL`/`IS NOT NULL` counterpart and explain any difference before presenting. For a predicate with = or <> on a nullable column, also count the rows where the column `IS NULL`, and state whether they are meant to be included.

Knowledge: `06_exemplars.sql` contains a NULL test written with IS NULL. The course database contains at least one employee whose `manager_id` is `NULL`, namely, the employee at the top of the hierarchy; otherwise, the baseline and the correct query both return no rows, and the comparison below passes trivially.

The comparison prescribed by the rule is what exposes the baseline predicate. With an additional tool call, Claude counts the rows that satisfy the candidate predicate and the rows that satisfy its `IS NULL` counterpart:

```
SELECT COUNT(*) FILTER (WHERE manager_id = NULL) AS candidate,
       COUNT(*) FILTER (WHERE manager_id IS NULL) AS is_null
```

```
FROM employees
```

The first count is 0 on any data, because `manager_id = NULL` is never true; the second is at least 1. The difference is the number of rows that the candidate silently drops. Since the rule requires every difference to be explained, Claude rewrites the predicate following the exemplar and executes it:

```
SELECT * FROM employees WHERE manager_id IS NULL;
```

With the rule in place, Claude normally writes `IS NULL` from the start; the comparison is the backstop for the case in which it drafts `= NULL` nonetheless.

J. Temporal-Logic Hallucination

Request:

"Show all orders placed in February 2024."

Baseline output:

```
SELECT * FROM orders
WHERE order_date BETWEEN '2024-02-01' AND '2024-02-28';
```

Issue: The model computes the last day of the month "by hand" and gets it wrong; 2024 is a leap year, so orders placed on 29 February are silently excluded. The query executes and returns a plausible set of rows. The DDL says that `order_date` is a `DATE`; it cannot say how many days a given month has. The same `BETWEEN` idiom fails in a further way on `TIMESTAMP` columns, where it cuts the last day at midnight; this is why the rule applies whatever the column type.

Mitigation:

Standing rule (Project instructions):

> Temporal ranges. Express every date or time range as a half-open interval, col >= start AND col < start of the next period (e.g., `order_date >= DATE '2024-02-01'` AND `order_date < DATE '2024-03-01'`), whatever the column type; never use `BETWEEN`, and never compute the last day of a period by hand. Take the type of the column (`DATE` or `TIMESTAMP`) from `01_schema.sql` and follow the pattern in `06_exemplars.sql`. Before presenting, count the selected rows a second time with a predicate that does not depend on the length of the period (`date_trunc` on the column) and check that the two counts coincide. When `07_reference_queries.sql` lists a boundary query for the period, run it and compare its result with the expected one recorded there.

Knowledge: `06_exemplars.sql` contains a half-open monthly range, while `07_reference_queries.sql` contains a boundary query for February 2024 together with its expected result. The course database contains orders dated 29 February and 1 March 2024, so that a range that stops one day early or runs one day late returns a different count.

Run-time check: Claude reads the type of `order_date (DATE)` in `01_schema.sql` and writes the range as a half-open interval:

```
SELECT * FROM orders
WHERE order_date >= DATE '2024-02-01' AND order_date < DATE '2024-03-01'
```

With an additional tool call, it counts the same orders through a formulation that does not depend on the length of the month, and checks that the two counts coincide:

```
SELECT COUNT(*) FROM orders
WHERE date_trunc('month', order_date) = DATE '2024-02-01';
```

Applied to the baseline, this comparison fails whenever an order was placed on 29 February. Finally, Claude runs the boundary query of `07_reference_queries.sql` and compares its result with the expected one.

K. Ambiguity-Induced Hallucination

The approach proposed by Saparina and Lapata [33] addresses linguistic ambiguity originating from the wording of natural-language questions themselves, distinguishing the following three forms:

- Scope ambiguity: A quantifier such as "each", "every", or "all" can be read two ways: collective, with one shared answer for the whole group, or distributive, with one answer per member of the group.
- Attachment ambiguity: A modifier placed after two joined items (`"X and Y [modifier]"`) can be read as applying to both items (broad) or to only one of them (narrow).
- Vagueness: When a word has more than one reasonable meaning and the schema does not say which is meant.

Category K is about the model silently picking one meaning and never telling the user that other meanings existed. Below, each of the three ambiguity forms receives its own worked example against the running PostgreSQL schema.

K.1 Scope ambiguity example

Request:

"Show the products every customer has ordered."

Two valid readings:

(a) Distributive: For each customer, list the products that customer ordered (one list per customer);

(b) Collective: List only the products that all customers have ordered in common (the intersection across every customer).

Baseline output:

```
SELECT c.customer_name, p.product_name
FROM customers c
     JOIN orders o ON o.customer_id = c.customer_id
     JOIN order_items oi ON oi.order_id = o.order_id
     JOIN products p ON p.product_id = oi.product_id;
```

Issue: The query silently picks the distributive reading (one row set per customer) and never mentions that the collective reading ("products common to every customer", which would need a `GROUP BY` product with a `HAVING COUNT(DISTINCT customer_id) = total customer count`) was also a valid answer to the same question. The query is syntactically fine and returns real rows, so nothing about the result looks wrong.

Mitigation:

Standing rule (Project instructions):

> Scope: treat "each", "every", "all" applied to a group as ambiguous between the distributive and the collective reading.

Knowledge: `06_exemplars.sql` contains a labeled pair of queries for a request with a quantifier: a distributive one (one row set per member of the group), and a collective one (relational division: `GROUP BY ... HAVING COUNT(DISTINCT ...)` equal to the size of the group).

Run-time check: Execution cannot reveal that a reading was chosen, since both translations run and return real rows. The loop is used instead to make the two readings concrete. Claude executes both and presents them side by side, each labeled and with its row count:

```
-- (a) distributive:  the products ordered by each customer
SELECT DISTINCT c.customer_name, p.product_name
FROM customers c
```

```
JOIN orders o ON o.customer_id = c.customer_id
JOIN order_items oi ON oi.order_id = o.order_id
JOIN products p ON p.product_id = oi.product_id;
-- (b) collective:  the products ordered by every customer
SELECT p.product_name
FROM products p
JOIN order_items oi ON oi.product_id = p.product_id
JOIN orders o ON o.order_id = oi.order_id
GROUP BY p.product_id, p.product_name
HAVING COUNT(DISTINCT o.customer_id) =
(SELECT COUNT(*) FROM customers);
```

Execution also supplies a fact that the wording cannot, that is, a probe for customers who have never ordered. If any exist, then reading (b) is empty by construction, which Claude states, so the empty result is not mistaken for a statement about the products. The ambiguity also has a didactic payoff. The collective reading is an instance of relational division, one of the constructs that students find hardest to write in SQL, so a request that a model would resolve silently becomes an occasion to introduce it. The comparison of the two readings also shows why the distributive query needs `DISTINCT`: without it, as in the baseline, a product that a customer bought in several orders is listed once per order.

K.2 Attachment ambiguity example

Request:

"Show customers and orders that are cancelled."

Two valid readings:

(a) Narrow attachment: "Canceled" applies only to "orders". Return every customer, and for each customer show only their orders that are canceled (customers with no canceled order may still appear, with no canceled-order rows).

(b) Broad attachment (as far as the schema allows): "Canceled" is meant to restrict the whole result to customer–order pairs where the order is canceled, so only customers who have at least one canceled order appear at all.

Baseline output:

```
SELECT c.customer_name, o.order_id, o.status
FROM customers c
     JOIN orders o ON o.customer_id = c.customer_id
WHERE o.status = 'cancelled';
```

Issue: An inner join combined with the `WHERE` clause silently applies the broad reading: customers with no canceled orders disappear from the result entirely. If the user meant the narrow reading (all customers, canceled orders only where they exist), the correct query would need a `LEFT JOIN` with the status filter moved into the join condition. The query runs without error and returns a plausible-looking table, so the wrong attachment choice is not visible from the output alone.

Mitigation:

Standing rule (Project instructions):

> Attachment: when a modifier follows two entities ("customers and orders that are cancelled"), check whether it applies to one or both.

Knowledge: `06_exemplars.sql` contains an exemplar of the narrow reading, an outer join with the filter on the inner table placed in the ON clause rather than in WHERE.

Run-time check: Claude executes both readings, and reports for each the number of distinct customers in the result; the difference is the number of customers without a canceled order, i.e., the rows on which the two readings disagree.

```
-- (a) narrow:  all customers, cancelled orders where they exist
SELECT c.customer_name, o.order_id, o.status
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.customer_id
       AND o.status = 'cancelled';
-- (b) broad:  only customers with a cancelled order
SELECT c.customer_name, o.order_id, o.status
FROM customers c
JOIN orders o ON o.customer_id = c.customer_id
WHERE o.status = 'cancelled';
```

If the difference is zero, the two readings coincide on the current data but not in general; Claude still states which reading it adopted. The value 'cancelled' is the one confirmed by the probe of Category F. The exemplar has a didactic value that goes beyond Category K: moving the filter on the inner table of a `LEFT JOIN` from the `ON` clause to the `WHERE` clause silently turns the outer join into an inner join, because the rows padded with NULLs fail the `WHERE` predicate. This is a classic novice error, and the two readings of the request show it as a pair of results that differ only in the customers without a canceled order.

K.3 Vagueness example

Request:

"Who are our top customers?"

Baseline output:

```
SELECT c.customer_name, COUNT(o.order_id) AS n_orders
FROM customers c
     JOIN orders o ON o.customer_id = c.customer_id
GROUP BY c.customer_name
ORDER BY n_orders DESC LIMIT 10;
```

Issue: In a business context, "top" could mean highest total revenue, highest revenue net of cancellations, most orders placed, or most recent activity. The model silently commits to one definition (for example, order count) when a different definition (total revenue) is very often what "top customer" means in practice. No SQL error occurs; the query answers a different business question than the one the requester had in mind. The baseline has two further defects, independent of the ambiguity. It groups by `customer_name`, which the schema does not declare `UNIQUE`, so two customers with the same name are merged into a single row. LIMIT 10 cuts arbitrarily among customers tied at the tenth position, so the list may change from one execution to the next.

Mitigation:

Standing rule (Project instructions):

> Vagueness: resolve business terms with `05_business_glossary.txt`; a term not in the glossary requires a clarification. In all three cases, never resolve silently: name the ambiguity, state the reading chosen, and either ask the user or present the alternative readings side by side, each labelled.

Knowledge: `05_business_glossary.txt` defines "top customers" as the customers ranked by revenue, i.e., `SUM(quantity × unit_price)` over the lines of their non-canceled orders, with a default list length of 10, including ties at the cut-off. If the instructor deliberately leaves the term out of the glossary, then the same rule makes Claude ask for clarification instead; thus, the same request supports two classroom demonstrations.

Run-time check: Claude applies the glossary definition, states it in the answer, and executes:

```
SELECT c.customer_id, c.customer_name,
```

```
        SUM(oi.quantity * oi.unit_price) AS revenue
FROM customers c
        JOIN orders o ON o.customer_id = c.customer_id
        JOIN order_items oi ON oi.order_id = o.order_id
WHERE o.status <> 'cancelled'
GROUP BY c.customer_id, c.customer_name
ORDER BY revenue DESC
FETCH FIRST 10 ROWS WITH TIES;
```

Grouping by `customer_id` and the clause `FETCH FIRST 10 ROWS WITH TIES` (PostgreSQL 13 and later), which keeps every customer tied at the cut-off, also remove the two defects of the baseline noted above. To show that the definition matters, Claude may also execute the alternative reading (number of orders) and report how many customers the two top-10 lists have in common.

Common thread across K.1–K.3 All three examples share the same root cause: the natural-language request has more than one correct SQL translation, the generated query is syntactically valid and returns real data under any reading, and nothing in the execution trace signals that a choice was made. The shared countermeasure is always the same shape: name the ambiguity type, state out loud which reading was chosen, and either ask the user or show the alternative readings rather than resolving the ambiguity silently.

A request for clarification is a final answer in the sense of Section 2.4 (τ_i = none): it closes the loop, and the user's reply is a new question answered by a new loop.

L. Hierarchical/Recursive-Structure Hallucination

Request:

"List every employee together with all of their indirect reports."

Baseline output:

```
SELECT e.first_name, m.first_name AS manager
FROM employees e JOIN employees m ON e.manager_id = m.employee_id;
```

Issue: This single self-join only recovers direct reports (one level of the hierarchy). The model hallucinates that a flat self-join satisfies a request that is inherently recursive over an arbitrary number of organizational levels, silently truncating the true result. The DDL does show the self-reference (`manager_id REFERENCES employees (employee_id)`), but it declares a single edge. How many levels the hierarchy has is a property of the data, and nothing in a `CREATE TABLE` statement says that a request about indirect reports requires following that edge repeatedly. The query executes and returns real rows, the direct pairs, which are a subset of the correct answer and look plausible.

Mitigation:

Standing rule (Project instructions):

> A self-referencing foreign key (flagged in `03_fk_graph.txt`) signals a hierarchy. For requests over "all levels", "indirect", "whole chain" or similar, write a `WITH RECURSIVE` query following the pattern in `06_exemplars.sql`, never a single self-join. Before presenting, run it on the reference sub-hierarchy and compare the result with the expected answer in `07_reference_queries.sql`.

Knowledge: `03_fk_graph.txt` flags `employees.manager_id` → `employees.employee_id` as a self-referencing edge, i.e., as a hierarchy; `06_exemplars.sql` contains a `WITH RECURSIVE` query on employees that returns every pair (manager, direct or indirect report) with its depth, which guards against cycles, while `07_reference_queries.sql` records the expected set of direct and indirect reports for one manager of a reference sub-hierarchy along with their depth. The course database contains a hierarchy at least three levels deep; otherwise, every report is direct and the flat self-join and recursive query both return the same pairs.

Run-time check: Claude recognises the self-referencing edge flagged in `03_fk_graph.txt`; following the exemplar, it writes:

```
WITH RECURSIVE chain (manager_id, report_id, depth) AS (
SELECT manager_id, employee_id, 1
FROM employees
WHERE manager_id IS NOT NULL
UNION ALL
SELECT c.manager_id, e.employee_id, c.depth + 1
FROM chain c
    JOIN employees e ON e.manager_id = c.report_id )
CYCLE report_id SET is_cycle USING path
SELECT m.employee_id, m.first_name, m.last_name,
    r.employee_id AS report_id, r.first_name AS report_first_name,
    r.last_name AS report_last_name, c.depth
FROM employees m
    LEFT JOIN chain c ON c.manager_id = m.employee_id AND NOT c.is_cycle
    LEFT JOIN employees r ON r.employee_id = c.report_id
ORDER BY m.employee_id, c.depth, r.employee_id;
```

The `CYCLE` clause (PostgreSQL 14 and later) is needed because the DDL does not prevent a cycle in `manager_id`, e.g., two employees each being recorded as the other's manager; without it, such data would make the recursion run without end. The depth column separates direct reports ($depth = 1$) from indirect ones ($depth \geq 2$). If the user means indirect reports only, the filter is $depth \geq 2$, and Claude states which reading it adopted, as for Category K. Claude then restricts the query to the manager recorded in `07_reference_queries.sql`, executes it, and compares the returned set with the expected one. Finally, it reports how many pairs have depth ≥ 2; these are the same rows omitted by the flat self-join of the baseline.

M. Explanation–Query Inconsistency Hallucination

Request:

"Show products that have never been ordered."

Baseline output:

```
-- Explanation given to the user:
-- ''This returns products with no matching row in order_items,
-- i.e., products that have never been sold.''
SELECT p.product_id, p.product_name
FROM products p JOIN order_items oi ON oi.product_id = p.product_id
WHERE oi.quantity = 0;
```

Issue: The prose explanation describes a correct anti-join strategy (`LEFT JOIN ... WHERE oi.product_id IS NULL`, or `NOT EXISTS`), but the generated SQL implements something entirely different (an inner join filtered on quantity = 0). The narrative and the executable artifact disagree, which is a distinct and dangerous hallucination mode because the fluent explanation lends unearned credibility to the wrong query. The DDL plays no part in this failure; the inconsistency is between two outputs of the model, each generated from the intended strategy, and nothing ties the explanation to the SQL text actually written. On the course database, where no order line has quantity 0, the query returns no rows. An empty result reads as a legitimate answer (every product has been ordered at least once), and the explanation makes it credible.

Mitigation:

Standing rule (Project instructions):

> Execute the query before explaining it. Derive the explanation from the SQL text actually executed, clause by clause, and from the rows it returned, never from the intended logic. Before presenting, compare the row count with an independent count of the same set (e.g., for products never ordered: the number of products minus the number of distinct products in `order_items`). When a reference query for the same request exists in `07_reference_queries.sql`, compare the two results before presenting the query or its explanation.

Knowledge: `07_reference_queries.sql` contains the anti-join written with `NOT EXISTS` for this request and its expected result. The course database contains at least one product that has never been ordered; otherwise, the correct query and the baseline both return no rows and the comparison passes trivially.

Run-time check: Claude executes the candidate query first; it returns no rows. Reading the executed SQL clause-by-clause, Claude finds that the inner join keeps only products that appear in `order_items`, i.e., products that have been ordered. Whatever the filter, this query cannot return a product that has never been ordered, which contradicts the explanation drafted with it. The independent count confirms the conflict:

```
SELECT (SELECT COUNT(*) FROM products) -
(SELECT COUNT(DISTINCT product_id) FROM order_items) AS never_ordered;
```

The result is greater than zero, while the candidate returned no rows. Claude rewrites the query as an anti-join:

```
SELECT p.product_id, p.product_name
FROM products p
WHERE NOT EXISTS
     (SELECT 1
     FROM order_items oi
     WHERE oi.product_id = p.product_id);
```

It executes this query, checks that its row count equals `never_ordered`, and compares its result with the reference query of `07_reference_queries.sql`. Only then does it write the explanation, from this query and the rows it returned.

Left to the user: Comparing the explanation with the query and the returned rows. Students must be taught that an explanation is evidence of correctness only when it describes a query that was executed and the rows that query returned.

N. Fabricated-Execution/Overconfident-Result Hallucination

Request:

"How many customers signed up in 2023?"

Baseline output:

```
-- Model response, no query actually executed:
‘‘Running this query returns 482 customers who signed up in 2023.’’
```

Issue: The model states a specific numeric result as though the query had been executed against the live database, when in fact no execution occurred and no such number can be known from the schema and prompt alone. This is hallucinated evidence of empirical fact, the most consequential category because it masquerades as verified ground truth rather than as generated code. In the baseline configuration, no database connection exists, so the failure is not that the query was not executed, which was impossible, but that the response claims it was. The DDL contains no data; no number of rows can be derived from it, and any number reported is an invention. The sentence has the form of a verified result and contains nothing to signal that it is not one.

Mitigation:

Standing rule (Project instructions):

> Never state a number, a row count or a row value that was not returned by a query executed with the MCP tool in this conversation. Show the executed query next to every reported result. If a query was not executed, say so explicitly and do not give a result.

Like the rule of Category A, this is one of the general standing rules of Section 2.2. It applies to every query, not only to requests of Category N.

Knowledge: The rule is self-contained. It constrains what Claude may state, not what it must know about the database.

Run-time check: Claude executes the query with the MCP tool and reports the number returned, with the query next to it:

```
SELECT COUNT(*)
FROM customers
WHERE signup_date >= DATE '2023-01-01' AND
      signup_date < DATE '2024-01-01';
```

The half-open range follows rule J. Before the final generation, Claude checks that every number in its answer was returned by a statement executed in the current loop. In the terms of Section 2.4, a number may appear in the final answer A_n only if it appears in a result R_i with i < n. In a conversation without a database connection, no R_i exists; the rule then makes Claude present the query as not executed and give no number. This is precisely the configuration in which the baseline of this category fabricates a number.

Left to the user: Applying the taught policy, i.e., for every number Claude reports, checking whether the query that produced it is shown, and asking for it where it is missing.

O. Security Assumption Hallucination

Request:

"Write a query to look up a customer by the name a user types in."

Baseline output:

```
-- Model comment:  ''This is safe because we only read data.''
query =
     ''SELECT * FROM customers
       WHERE customer_name = '' + userInput + ''';
```

Issue: The model builds a query by direct string concatenation and asserts, incorrectly, that read-only access makes this safe. This is a hallucination of a nonexistent security guarantee, as unsanitized concatenation is SQL-injectable and can be exploited even in a read-only context to exfiltrate arbitrary data via `UNION`-based injection. The DDL does not prevent this; it describes the data, not how an application passes user input to a query, and `customer_name VARCHAR(100)` says nothing about who supplies the value compared with it. Execution with an ordinary value also does not reveal the flaw: with a name such as "Rossi", the concatenated query runs and returns the correct row.

Mitigation:

Standing rule (Project instructions):

> For any query template that will receive user input, use parameterized statements only (e.g., `WHERE customer_name = $1`); never concatenate user input into SQL text; assume a least-privilege, read-only database role. Check every template against these three points before presenting it, whatever your own assessment of its safety; read-only access does not prevent injection. Test the template by executing it with a legitimate value that contains an apostrophe (e.g., O'Brien), following the pattern in `06_exemplars.sql`.

Knowledge: `06_exemplars.sql` contains a parameterized template, both in SQL (`PREPARE ... EXECUTE`) and in the placeholder form of the host language used in the course (e.g., %s with psycopg in Python). The course database contains at least one customer whose name contains an apostrophe.

Run-time check: Claude instantiates the baseline template with the value O'Brien and executes the resulting statement. The apostrophe closes the string literal early, and PostgreSQL rejects the statement with a syntax error. Thus, a legitimate name breaks the query by the same mechanism that a malicious input would exploit. Claude then writes the parameterized form and executes it with the same value:

```
PREPARE find_customer(VARCHAR) AS
SELECT customer_id, customer_name, email, country
FROM customers
WHERE customer_name = $1;
EXECUTE find_customer('O'Brien');
```

The statement returns the customer; the value reaches the engine as data, never as SQL text (in the SQL literal of `EXECUTE`, the apostrophe is doubled; in the host language, the value is passed unchanged). Finally, Claude checks the template against the three points of the rule, and states (against the claim of the baseline) that a read-only role limits what an injected statement can change rather than what it can read.

3.1. Rationale for Candidate Merging

Following the criterion of Section 2.5, candidates were merged when they shared both the root cause and the countermeasure. The merged candidates are the following:

- Category B merges two candidates: a column absent from the schema, and a derived value treated as a stored column (e.g., an order total). The root cause is the same, a column taken from schemas seen in training rather than from the declared one, and so is the countermeasure: the column catalog, which marks every attribute as STORED or DERIVED, and the check against `information_schema.columns`.
- Category G merges functions and syntax of other DBMSs (e.g., `DATE_FORMAT` and `IFNULL` from MySQL). The root cause is the same, a target dialect that the DDL does not state, and so is the countermeasure: a rule stating the dialect, and the engine error read within the loop.
- Category J merges two boundary errors: a last day of the period computed by hand (e.g., 28 February in a leap year), and `BETWEEN` on a `TIMESTAMP` column, which cuts the last day at midnight. The root cause is the same, a range written as a closed interval, and so is the countermeasure: the half-open interval.
- Category K merges three forms of ambiguity: scope, attachment, and vagueness. Their root cause is the same, a request with more than one valid translation, resolved silently; so is their countermeasure: name the ambiguity, state the reading chosen, and either ask the user or show the alternative readings.
- Category N merges fabricated execution, i.e., a result reported as if a query had been run, and overconfident results, i.e., a value stated as certain without a query that returned it. The root cause is the same, a number not grounded in an execution; so is the countermeasure: the provenance rule.

3.2. Rationale for Candidate Splitting

Following the criterion of Section 2.5, candidates that share a symptom or a locus but differ in mechanism were kept separate. The split candidates are the following:

- Categories C, D, and E all concern joins and aggregation, but differ in their mechanisms and countermeasures:

  - C is a relationship that does not exist. It is prevented by the foreign-key graph and detected by checking every join predicate against the declared foreign keys.
  - D is valid relationships combined so as to multiply rows; it is handled by one CTE per branch, a `COUNT(*)` vs. `COUNT(DISTINCT)` check on the combined result, and a comparison of each metric with its own branch.
  - E is a single one-to-many relationship aggregated at the wrong grain, with no row duplicated. It is handled by a two-stage aggregation.

- A and B, which are grouped by [17] under "Non-Existent Schema", are kept apart here because their countermeasures lead to different outcomes. For A, the correct answer is often a refusal: no declared table holds the data, and the rule makes "this question cannot be answered from this database" an admissible answer. For B, the requested value usually exists in derived form, and must be computed with the derivation recorded in the column catalog, a knowledge file that A does not need.
- G and H both misuse SQL constructs, but G raises an engine error that the loop catches, whereas H executes silently and returns a well-typed number. Therefore, H needs a rule on the operand types of a division and a check of the expression's type rather than of its value.
- F and K.3 both involve a request term that does not match the data, but differ in what the term refers to. In F, the term stands for a value of a column ('completed'). The `SELECT DISTINCT` probe shows that the value does not exist, and the term is then mapped to a stored value. In K.3, the term stands for a criterion ("top"), with no column to probe. No execution can reveal that a choice was made, and the definition can come only from the glossary or the user.
- D and L both use the self-reference of employees, but D combines it with a second branch at one level, whereas L follows it over an arbitrary number of levels. D is handled by per-branch aggregation and L by a recursive query.
- M and N both concern how results are reported. M is an inconsistency between prose and query (execute before explaining), while N is a claim of an execution that never occurred (provenance rule).

4. Related Work

This section carries out the cross-check mentioned in Section 1.3. It positions the fifteen categories against four works that classify hallucinations or errors in LLM output: the two-class taxonomy of Qu et al. [16], the fine-grained taxonomies of Shen et al. [17] and Liu et al. [32], and the domain-general hallucination survey of Huang et al. [34]. The first three are specific to text-to-SQL; the fourth supplies the general vocabulary (factuality vs. faithfulness hallucination) within which text-to-SQL failures can be placed. Because the proposed taxonomy is expert-driven and illustrative (Section 2.5), the comparison can establish two things: whether each category has a counterpart in taxonomies derived from large sets of failed queries, and where it has none, whether the absence is accidental or follows from how those taxonomies are built. It cannot establish how often a category occurs, a question that is outside the scope of this paper (Section 1.3).

The comparison proceeds at two levels. Section 4.1 compares whole works, which Table 3 classifies by objective, application domain, and type of contribution. Table 4 uses five dimensions that emerged from reading the works and that capture how each of them treats hallucination: (i) the error framing and its granularity, (ii) the unit of analysis, (iii) the empirical basis, (iv) the point at which mitigation acts (before, during, or after generation), and (v) the pedagogical orientation. In both tables, the last row places the present paper on the same dimensions. Sections 4.2 and 4.3 examine the previous works one at a time. Section 4.4 compares individual categories (Table 5). Each row is one of the

fifteen categories, mapped onto the corresponding class of [16,17,32,34]. The work-level view explains why differences arise, while the category-level view shows which categories they produce.

Table 3. Works classified by objective, application domain, and type.

Ref.	Objective	Application Domain	Type
[16]	Identify the hallucination types arising at each stage of the two-stage (schema linking, logical synthesis) in-context-learning text-to-SQL pipeline, and mitigate them through Task Alignment (TA-SQL framework)	Cross-domain text-to-SQL benchmarks (BIRD, Spider family)	Methodological + experimental
[17]	First comprehensive empirical study of errors made by ICL-based text-to-SQL techniques; propose an error detection-and-repair framework (MapleDoctor)	Cross-domain text-to-SQL benchmarks (Spider, BIRD)	Empirical study + methodological (tool)
[32]	Provide a benchmark for detecting and categorizing semantic errors in NL2SQL output, and assess LLMs as error detectors	Cross-domain text-to-SQL benchmarks (BIRD, Spider)	Benchmark/dataset + experimental
[34]	Survey hallucination in LLMs: taxonomy, causes, detection, benchmarks, mitigation, limits of retrieval augmentation	Domain-general natural-language generation and information retrieval (not SQL-specific)	Survey (theoretical/conceptual)
This paper	Make the variety of text-to-SQL hallucinations visible to SQL learners, and give instructors a checklist that pairs each category with a mitigation assigned to instructor, agentic loop or user	University database education (Master's level); five-table PostgreSQL teaching schema	Conceptual and illustrative (expert-driven taxonomy + mitigation framework, with worked examples)

Table 4. Works classified by hallucination framing, unit of analysis, empirical basis, mitigation locus, and pedagogical orientation.

Ref.	Framing (Granularity)	Unit of Analysis/Setting	Empirical Basis	Mitigation Locus	Pedagogical Orientation
[16]	Two classes (schema-based, logic-based). Six sub-types: Schema Contradiction, Attribute Overanalysis, Value Misrepresentation, Join Redundancy, Clause Abuse, Mathematical Delusion	Single NL-to-SQL generation, scored offline against a gold query	BIRD dev and other mainstream benchmarks; several LLMs including GPT-4	During generation (task-aligned prompting in both stages)	No
[17]	Seven categories for 27 error types (syntax, schema, logic, convention, semantic, not-an-error, others), derived by open card sorting of 4602 incorrect queries	Finished SQL query (output of the generation step) analyzed post hoc against the gold result	Spider and BIRD dev; four ICL techniques (MAC-SQL, DIN-SQL, CHESS, DEA-SQL); GPT-3.5, GPT-4o; open-source LLMs in the generalizability study	After generation: rule-first detection and repair (DBMS errors, AST anti-patterns, schema checks, data probes); LLM repair as fallback	No

Table 4. *Cont.*

Ref.	Framing (Granularity)	Unit of Analysis/Setting	Empirical Basis	Mitigation Locus	Pedagogical Orientation
[32]	Nine main categories, 31 subcategories of semantic errors, organized by SQL component; syntax errors excluded	(NL, DB, SQL) triple judged against a gold query	2018 instances (999 incorrect), labelled by two annotators (Cohen's k = 0.78); LLMs evaluated as detectors (average accuracy about 75%)	After generation (detection, prerequisite for correction)	No
[34]	Factuality vs. faithfulness hallucination, with sub-types; causes traced to data, training and inference	Not applicable (literature synthesis)	Literature	Catalog of data-, training- and inference-level mitigation	No
This paper	Fifteen categories at an intermediate granularity; eleven within or near the two classes of [16], four (K, M–O) outside both	The loop of generations answering one question within a session (Section 2.4); its output includes the answer shown to the user, not only the SQL; queries executed on a live PostgreSQL database via MCP	Literature, 27 years of teaching; one worked example per category, with a constructed baseline and a mitigated configuration run with Claude Sonnet 5 (Section 2.5); no frequency estimates	Before generation (context pack: standing rules and knowledge files); during the loop (rule-prescribed run-time checks on the live database); user, for F, K, M, N	Yes (primary aim)

4.1. Work-Level Comparison

Table 5 maps each category onto the four classifications.

Table 5. Mapping of the fifteen categories onto existing classifications. IDs and names in the Shen column follow [17]; names in the Liu column follow Figure 3 of [32]; names in parentheses in the Qu column are subtypes of [16]. † = assignment made by us on the basis of the definitions in the cited work. NA = not applicable. The last column states whether the category has a counterpart in at least one of [16,17,32].

Cat.	Qu et al. [16]	Shen et al. [17]	Liu et al. [32]	Huang et al. [34]	Static Counterpart
A	schema-based (Schema Contradiction) †	B1 Non-Existent Schema	syntax error (outside the 31 subcategories)	faithfulness (context inconsistency) †	Yes
B	schema-based (Schema Contradiction) †	B1 Non-Existent Schema	syntax error (outside the 31 subcategories)	faithfulness (context inconsistency) †	Yes
C	schema-based (Schema Contradiction) †	D3 Comparing Unrelated Columns; E9 Comparing Wrong Columns (related)	Join Condition Mismatch	faithfulness (context inconsistency) †	Yes
D	logic-based †	no dedicated type (symptoms: E6 Wrong COUNT Object, E8 Missing DISTINCT)	no dedicated subcategory	outside the dichotomy (cause: reasoning failure) †	No dedicated category; observable as result mismatch
E	logic-based (Clause Abuse, partial) †	E5 Unaligned Aggregation Structure	Aggregate Functions; Clause Missing (partial); Subquery Missing (partial)	faithfulness (instruction inconsistency) †	Yes
F	schema-based (Value Misrepresentation)	D1 Violating Value Specification	Value Mismatch	factuality (factual contradiction)	Yes

Table 5. *Cont.*

Cat.	Qu et al. [16]	Shen et al. [17]	Liu et al. [32]	Huang et al. [34]	Static Counterpart
G	logic-based (Mathematical Delusion: invented functions) †	A1 Function Hallucination	syntax error (outside the 31 subcategories)	outside the dichotomy	Yes
H	logic-based (Mathematical Delusion, partial) †	C1 Implicit Type Conversion	Data Format Mismatch; Conversion Functions (missing cast)	outside the dichotomy	Yes
I	logic-based †	C3 Ascending Sort with NULL (partial)	Comparison Operator	outside the dichotomy	Yes
J	logic-based †	E3 Improper Condition (partial)	Date/Time Functions; Comparison Operator; Value Mismatch (partial)†	outside the dichotomy	Yes
K	NA (outside both classes)	F3 Output Ambiguity, filed as "Not an Error"	NA (ambiguous questions removed at curation)	faithfulness (instruction inconsistency †)	No (not representable in a static triple)
L	logic-based †	NA	NA	faithfulness (instruction inconsistency †)	No dedicated category; observable as result mismatch
M	NA	NA	NA	faithfulness (logical inconsistency)	No (not representable in a static triple)
N	NA	NA	NA	factuality (factual fabrication)	No (not representable in a static triple)
O	NA	NA	NA	factuality (factual contradiction) †	No (not representable in a static triple)

4.2. Text-to-SQL Error Taxonomies

Qu et al. [16] split text-to-SQL hallucinations into schema-based hallucinations, which arise during schema linking (Schema Contradiction, Attribute Overanalysis, Value Misrepresentation), and logic-based hallucinations, which arise during logical synthesis when the schema elements are correct but the SQL logic is not (Join Redundancy, Clause Abuse, Mathematical Delusion). Eleven of the fifteen categories fall within or near these two classes (Table 5). Value Misrepresentation, an imagined format of a stored value, is the direct precedent of Category F, and Mathematical Delusion, which includes invented functions, is the closest precedent of Category G. The four remaining categories (K, M–O) fall outside both classes, as [16] defined hallucination types on the generated SQL query, whereas these four concern the answer returned to the user. Conversely, Attribute Overanalysis (an unrequested column in the projection) has no counterpart here, since surface-level result-shaping errors are excluded (Section 2.5). In [17,32], this base taxonomy is refined to a granularity closer to ours.

Shen et al. [17] analyzed 4602 incorrect queries produced by four ICL-based techniques on the Spider and BIRD development sets and organized them by open card sorting into 27 error types across seven categories: (a) Syntax Error (three types): the query fails to parse; (b) Schema Error (four types): the query parses but fails schema resolution; (c) Logic Error (three types): the flaw is evident without knowing the database or the question; (d) Convention Error (three types): the flaw is detectable from the schema alone; (e) Semantic Error (nine types): the flaw is detectable only by knowing both schema and question; (f) Not an Error (three types): gold-query errors, foreign-key violations in the data, and ambiguous questions; (g) Others: unclassifiable and miscellaneous errors. Seven of the

fifteen categories (A–C, E–H) have a direct or close counterpart in [17] (Table 5), and two (I, J) have a partial one. Three examples are telling: Category F corresponds to D1 Violating Value Specification, Category G corresponds to A1 Function Hallucination, examples of which include a MySQL function invoked against SQLite, and Category H corresponds to C1 Implicit Type Conversion, where the example is the same (integer division that truncates a ratio). This convergence between categories identified on a five-table teaching schema and types derived from thousands of benchmark failures is consistent with the view, as stated in Section 1, that the categories are properties of schema-grounded text-to-SQL generation rather than artifacts of one schema. In [17], it was reported that for some types with strong logic features, a more capable model does not reduce the number of errors, which is in line with the model-agnostic reading of the categories. The types of [17] without a counterpart here are those excluded in Section 2.5: syntax slips (A2, A3), result-shaping errors (E2, E7), using = instead of IN (C2), and benchmark artifacts (F1, F2).

The convergence extends to mitigation. Several detection strategies of MapleDoctor, the repair framework that [17] derives from its taxonomy, coincide with run-time checks of Section 3: a join predicate with columns that are not linked by a foreign key is flagged (Category C); the stored values of an enumeration-style column are probed (F); an integer division without a cast is detected in the parse tree (H); and the error message of the engine is read for non-existent functions and schema elements (A, B, G). In [17], it was further reported that repair methods provided with execution feedback fix more syntax and schema errors than methods that only ask the model to regenerate, and that regenerating every query without targeted information introduces new errors. Feeding execution results back to the model is not new, as several of the techniques studied in [17] already do this; the loop in Section 2.3 follows the same principle [40]. What differs is who runs the checks and for whom; MapleDoctor is an automated post-generation component of a benchmark pipeline, while here the same kinds of checks are standing rules of a context pack, run by the assistant within a session and visible to the student who reads the answer.

Liu et al. [32] introduced NL2SQL-BUGs, a 2018-instance benchmark with a two-level taxonomy of semantic errors (nine main categories, 31 subcategories). The taxonomy is organized by the SQL component in which the error lies (attributes, tables, values, operators, conditions, functions, clauses, subqueries, other). Each instance is a triple (NL, DB, SQL); its correctness label is obtained by comparing the execution result with that of a gold query, and its error types are assigned by two annotators. In [32] syntax errors (i.e., queries that fail to execute, including those with invalid references to tables or columns) were separated from semantic errors, with the latter found to account for about 99% of all errors made by CodeS on Spider and BIRD. Several of our categories map onto subcategories of [32]: Cat. C (fabricated join path) corresponds closely to Join Condition Mismatch, Cat. F (enumerated-value hallucination) to Value Mismatch, Cat. I (NULL semantics) to Comparison Operator, Cat. H (integer division) to Conversion Functions (through the missing cast), and Cat. E (aggregation scope) partially to Aggregate Functions, Clause Missing, and Subquery Missing, the last describing the missing first stage of aggregation.

However, the two taxonomies diverge in what they are taxonomies of: NL2SQL-BUGs classifies a static already-generated SQL artifact against a fixed gold query, and its unit of analysis is the triple (NL, DB, SQL) evaluated post hoc. Several of our categories fall outside that unit:

- Cat. A, B, and G (non-existent table; non-existent column; function of another DBMS) would be classified under [32] as syntax errors, outside its 31 subcategories. They are retained here for two reasons: first, they are the only categories in which executing the baseline query suffices to expose the error (Table 2), which makes them the natural first illustration of the loop for students; second, even for them, the engine error does

not determine the correct outcome. For Category A, the correct answer is often that the database cannot answer the question, an outcome that only the standing rule makes admissible.

- Cat. D (cardinality/fan-out) has no counterpart among the 31 subcategories. A fan-out query joins only declared relationships with correct join conditions, and both joined tables are needed, so neither Join Condition Mismatch nor Table Redundancy applies. Its wrong totals would be detected as a result mismatch, but would have to be filed under a generic subcategory such as Aggregate Functions, which does not separate them from Category E.
- Cat. L (hierarchical/recursive structure) likewise has no counterpart: in [32], there is no subcategory for recursive traversal, and the flat self-join of the baseline returns a subset of the correct rows that would also be filed under a generic subcategory.
- Cat. K (linguistic ambiguities) presupposes an NL request with more than one legitimate SQL interpretation. In [32], such requests were removed during curation, since the correctness labels in that work are defined relative to a single ground-truth query per instance.
- Cat. M, N, and O (explanation–query inconsistency, fabricated execution, security assumption hallucination) fall entirely outside the scope of NL2SQL-BUGs, since its (NL, DB, SQL) triple contains no natural-language explanations, claims about whether execution occurred, or statements about security.

One finding of [32] bears directly on the design of the loop. When asked whether a given query is semantically correct, the strongest models tested (GPT-4o, Claude-3.5-Sonnet) reached about 76–77% accuracy (Table 1 of [32]);, while when asked which type of error a query contains, they stay below 50% for most categories (Figure 6 of [32]). This is independent benchmark-scale evidence that an LLM's unaided judgment of SQL correctness is unreliable. Together with the repair results of [17] recalled above, it supports grounding verification in execution and the rule-prescribed checks of Section 2.3 rather than in the model's own confidence or explanation (cf. Cats. M and N). In summary, NL2SQL-BUGs and the present taxonomy are complementary: the fine-grained classification in [32] covering what kind of semantic mistake a finished SQL query contains is useful for benchmarking and for training error detectors, while the taxonomy proposed here classifies failures of the loop that answers a question, including the answer reported to the user, and pairs each failure with a countermeasure.

A recent taxonomy follows the same design. Lin et al. [31] used open coding to annotate 3968 incorrect queries generated by two LLMs on Spider, then organized them into 25 types in four groups: syntax errors, semantic errors which the engine reports at run time (e.g., a non-existent table or an invalid use of a function), intent-violating hallucinations, which execute without error but do not answer the question, and other errors. The unit of analysis was that of [17], i.e., a finished query labeled against a gold result, and to our reading it has no counterpart for Categories K or M-O; for this reason, it is not added to Tables 3–5. One of its findings does bear directly on Table 2: intent-violating hallucinations, i.e., failures that execution does not report, made up about 78% of the annotated errors, and the most frequent of them concerned join logic. Therefore, the engine error that exposes Categories A, B, and G is the exception rather than the rule, and the checks prescribed by the standing rules provide what the loop needs for the remaining categories.

4.3. Domain-General Hallucination Taxonomy

The survey by Huang et al. [34] proposes a domain-general split between factuality hallucinations, i.e., discrepancy between generated content and verifiable real-world facts

(factual contradiction, factual fabrication) and faithfulness hallucination, i.e., content deviating from the user's instruction, the supplied context, or its own internal logic (instruction in the former, context and logical inconsistency in the latter). In schema-grounded SQL generation, the two branches can be separated by a simple criterion. What the model receives in its context, i.e., the DDL in both configurations of Section 2.5 and the context pack in the mitigated one, is context; contradicting it is a faithfulness hallucination. What is stored only in the database, such as the values of a column or the number of rows, is fact, verifiable by a query.

Under this criterion, Categories A–C are context inconsistencies: the DDL in the conversation declares every table, column, and foreign key, but the model writes something else. Without the DDL in context, they would be factual fabrications; this is not the setting of Section 3. Category F is a factual contradiction, since the domain of status is data, not schema. Category N is a factual fabrication and the cleanest match in the whole taxonomy, that is, a numeric claim with no grounding at all, unverifiable by construction. Category O is a factual contradiction as well; the hallucinated content is the assurance that read-only access makes a concatenated template safe, a claim contradicted by established knowledge about SQL injection, while the injectable template is the defect that the assurance conceals. On the faithfulness side, Category M is a near-textbook case of logical inconsistency, and Categories K, E, and L read as instruction inconsistency (an ambiguous request resolved silently, an average order value computed as an average line value, and a request over all levels answered over one). Categories D and G–J fall outside the dichotomy: D results from a reasoning failure over relationships that the context states correctly, and G–J violate the dialect or the semantics of SQL rather than any instruction, context, or fact.

The correspondence is more informative on causes. In [34] hallucinations are traced to data (e.g., knowledge boundaries), training (pretraining, supervised fine-tuning, reinforcement learning from human feedback), and inference (e.g., overconfidence, reasoning failure). Category F and the target dialect behind Category G are knowledge boundaries specific to one database instance and one engine; no model capability compensates for data the model has never seen, which is why the mitigation is a probe of the data and a stated dialect rather than of a stronger model. By contrast, categories A and B occur even though the relevant information is in context. They matches what was described in [34] as overconfidence (i.e., favoring a fluent continuation over the supplied context) and conflicts between parametric and contextual knowledge. Here, it involves schemas seen in training as opposed to the declared one. Categories A, K, and N also match a training-related cause: supervised fine-tuning teaches models to complete every response rather than to express uncertainty or decline, so a question that the database cannot answer, a reading that cannot be chosen, or a number that cannot be known still receives a fluent answer. Categories D, E, and L match the reasoning failures of [34]: the information needed is present, but the multi-step relational reasoning over it fails.

On mitigation, the countermeasures of Section 3 act at inference time and leave the model unchanged. In the organization of [34], the context pack corresponds to retrieval augmentation in its simplest form, in which external knowledge is placed in the prompt once prior to generation; the execute–observe–correct loop corresponds to post hoc verification and revision, in which a generated answer is checked against external evidence and revised. One property distinguishes text-to-SQL from most tasks surveyed in [34], namely, that for every claim which can be expressed as a query, the database is an external oracle that returns deterministic evidence, whereas open-ended generation must approximate factuality through retrieved documents. Table 2 lays out the limits of this oracle: it reports an error by itself only for Categories A, B, and G; for the other twelve it answers only the checks that the standing rules prescribe, while no execution can replace a human decision

for Category K and the user parts of Categories F, M, and N. This is why these categories retain disambiguation and provenance practices, closer to the clarification-seeking strategies reviewed in [34]. In the end, [34] is a domain-general survey with no code generation case study, schema-grounding mechanics, or no pedagogical framing, so the present taxonomy is not a competing classification of hallucination in the abstract but a domain-specific instantiation that exploits the one property—executability—that a survey of open-ended text cannot assume.

4.4. Category-Level Mapping and Contribution

Table 5 maps each of the fifteen categories onto [16,17,32,34]. Eight categories (A–C, E–I) have a direct or close counterpart in at least one of the text-to-SQL taxonomies, and one (J) has a partial one. Two (D, L) have no dedicated counterparts, although their effects are observable in a static evaluation as a result mismatch; what is new for D is its separation from aggregation scope errors (Category E), while for L what is new is its recognition as a distinct failure of recursive traversal. The remaining four categories (K, M–O) have no counterparts, and cannot have them in a static evaluation for the reason given in the first of the three differences below, which separate the present proposal from [17,32].

- *Unit of analysis.* In [17,32], finished SQL queries were labeled against a gold result, i.e., a triple (NL, DB, SQL). The unit here is the loop that answers one question (Section 2.4), the last generation of which produces the answer A_n shown to the user (text, numbers, and explanations as well as SQL). Categories K, M, N, and O concern A_n: whether it discloses that a reading was chosen, whether its explanation describes the query executed, whether its numbers were returned by an executed query, and whether its assurances are true. None of these is represented in a triple. Ambiguity as such can be represented statically, as AMBROSIA [33] does by pairing an ambiguous question with several gold queries, but a static evaluation cannot represent whether the answer discloses that a choice was made. In [17], ambiguous questions were filed under "Not an Error" (F3, Output Ambiguity), while in [32] they were removed during curation.
- *Derivation*. Shen et al. [17] built their taxonomy by open-card sorting of 4602 incorrect queries by three co-authors. Liu et al. [32] organized theirs by SQL component, and had two annotators label the benchmark instances. Ours is expert-driven and illustrative, and as such we make no claim about the frequency of any category.
- *Purpose*. The previous works support diagnosis, benchmarking, and automatic repair, in the case of [17] on SQLite-based benchmarks. Our work pairs each category with a countermeasure that is assigned either to the instructor (I), to the execute–observe–correct loop (L), or for certain categories (F, K, M, N) to the user (U) (Table 2).

In summary, the field has a coarse taxonomy with two classes [16] and three fine-grained ones with 27 types [17], 31 subcategories [32], and 25 types [31] built for diagnosis and benchmarking. Our proposed taxonomy sits between them in granularity, converges with them where their unit of analysis allows, and adds four categories that become observable only when the unit includes the answer reported to the user. To our knowledge, none of the existing taxonomies is built for teaching SQL or assigns the mitigation of each category to the instructor, loop, or user; this assignment, rather than the categories alone, is what our proposal contributes to the literature reviewed above.

5. Discussion

This section discusses the proposal from the angles that Sections 3 and 4 leave open. Section 5.1 relates it to classroom experiences that also use LLM failures as teaching material. Section 5.2 reads the fifteen mitigations of Section 3 by actor. Section 5.3 explains why the

categories are expected to hold beyond any specific model, whereas their frequency is not. Section 5.4 discusses what the zero-cost requirement implies for the model tier and the size of the context pack, and Section 5.5 draws from it the scope of the proposal. Finally, Section 5.6 lays out the limitations of this study.

5.1. Pedagogical Positioning

Farinetti and Cagliero [41] report the following classroom experience. In two undergraduate database courses (166 students, B.S. Computer Engineering and Management Engineering), teams invented SQL exercises on given schemas, prompted ChatGPT 3.5 Turbo with each one, tried to make it produce a wrong solution, and commented on its output. Teacher annotation of the 142 exercises showed that ChatGPT's solution was fully correct in only 28% of cases. Participation was associated with better results on the SQL questions of the final exam. Like the present paper, LLM failures are treated as teaching material in [41], and the aim is to develop a critical attitude toward the tool. Two of their observations bear directly on our categories. Among the exercise texts that the teachers judged unclear or misleading, 8.5% asked for information that the database did not hold and 21.3% needed context that the schema did not provide, such as "best-seller" books or films produced in "the last week of October" (Table 1 of [41]). The first case is the situation of Category A, to which the correct answer is that the question cannot be answered from the database; the second is that of Category K.3, and for period boundaries, of Category J. Likewise, several labels used for errors of ChatGPT in [41] (tables or attributes, aggregate functions, GROUP BY, IS NULL) fall within our Categories A–C, D–E, and I.

The above work differs from ours in three respects. First, its categories are labels assigned after the fact to the errors observed in a chat without access to the database, whereas ours are defined in advance and each is paired with a check and a countermeasure. Second, the student is the only safeguard during the activity, whereas our framework also assigns mitigation to the instructor (context pack) and the agentic loop (execution against a live database) (Table 2). Third, neither execution nor the reporting stage of an agentic session were considered in [41], so failures of the kind captured by our categories M–O are outside its scope. Our taxonomy can be read as a structured SQL-specific instrument for exercising the critical attitude that [41] fosters. It also offers a way to pursue two directions that were left to future work in [41], namely, directing student exploration towards specific categories of failure and making students reflect on how the prompt affects the model's solutions. In our setting, the second direction becomes a concrete exercise: the same request is put to an assistant without and with the context pack, and students compare the answers they actually obtain with each other and with the illustrative baseline of Section 3.

Ponzini et al. [19] took a complementary perspective. Their framework logs and categorizes the errors of students rather than those of a model, then uses generative AI as a tutor that guides students through the resolution of their own errors without giving them the solution. The two perspectives meet in Section 1.1, where several errors that students make year after year correspond to categories of the present taxonomy, so the same misconception can be met first in one's own query and then in the model's.

5.2. The Mitigations Read by Actor

Section 3 pairs each category with a standing rule, the knowledge files the rule points to, and a check run within the loop. Read across the fifteen categories, these mitigations reduce to the contributions of the three actors in Table 2, which together make up the mitigated configuration of Section 2.5. This reading shows how the work is divided, which Section 4.4 identifies as a contribution of our proposal.

- *Instructor* (all fifteen categories). The instructor writes the standing rules and the seven knowledge files one time; since the DDL is present in both configurations, what the pack adds is exactly what Section 3 found to be missing. Three rules are self-contained (G, H, N): they prescribe a behavior, namely, the target dialect, operand types of a division, and provenance of every number, that no file about the database could supply. The other twelve point to files: the schema (A, J), column catalog (B, E), foreign-key graph (C, D, L), value catalog (F), business glossary (E, F, K), exemplars (I, J, K, L, O), and reference queries (J, L, M).
- *Loop* (all fifteen categories). Execution alone exposes three categories (A, B, G) through the error that the engine reports. For the other twelve, the loop is effective only through the checks that the rules prescribe. These are of three kinds: checks on the system catalog, which examine the structure of a query rather than its result (B, C, H); checks on the data, which compare the result with a probe, an independent count, or a reference query (D, E, F, I, J, L, M); and checks that make visible what is hidden by a single execution by executing every reading of an ambiguous request (K) or a test input (O). For N, the check is that every number in the final answer was returned by a statement executed in the same loop.
- *User* (four categories). Part of the mitigation remains with whoever is interacting with Claude: deciding the meaning of a business term that the glossary does not define (F), choosing among the readings of an ambiguous request (K), comparing an explanation with the executed query and its rows (M), and checking that every reported number comes with the query that produced it (N). These are the habits that our course aims to leave with students for times when no context pack is available.

The checks of the loop can also be read in terms of metamorphic testing, which Lin et al. [31] applied to text-to-SQL. Their method detects intent-violating hallucinations without a gold query: it rewrites the question into several variants, each linked to an expected relation between results (equality, inclusion or containment), asks the model to write a query for each variant, executes all of them on the same database, and flags the original query when the proportion of violated relations exceeds a threshold. Several checks of Section 3 are relations of the same kind, written by hand: the two counts of Category J must coincide, as must the row count of Category M and the independent count of the same set, and each metric of Category D and the same metric computed from its own branch alone. Two differences follow from the purpose of the present proposal. First, in [31] the relations are selected automatically from a knowledge base of past failures, and the verdict rests on a majority vote. Here, each relation is fixed in advance by the instructor for one category and named in a standing rule, so a student reading the answer can see which check was run and why. Second, in [31], the question was varied while letting the model write a new query for each variant, whereas the checks of Section 3 vary the query or compute the same set another way while adding one or two statements per category. In [31], a result was reported that agrees with that of [32], recalled in Section 4.2: when asked to judge its own queries, the same model missed most of its hallucinations (for GPT-4o on Spider, a recall of 0.07), whereas the comparison of executed results detected many more. Here, verification grounded in execution is the principle on which the loop of Section 2.3 rests, as opposed to relying on the model's own assessment.

5.3. Model-Agnostic Categories, Model-Dependent Frequency

The proposed taxonomy is expected to be model-agnostic in its categories but model-dependent in frequency and severity. These two claims rest on different grounds, and each has a different status.

The first follows from where the categories come from: each arises from something that the schema DDL does not provide (Section 3), from a rule of behavior, a fact about the data or the business, or evidence that only execution can provide. For example, Category B exists because whether a value such as an order total is stored or derived is a design decision that a column name does not reveal. Category C exists because a join predicate can be written between any two type-compatible columns, whether or not the DDL declares them as a relationship. Category D exists because SQL's join semantics multiply rows by design: any model, however capable, inflates the totals if it joins two one-to-many branches before aggregating them. Category F exists because the values stored in a column are data, not schema (Section 4.3). A model with a different training history meets the same gaps, and the countermeasures of Section 3 fill them from outside the model.

Three independent findings are consistent with this reading. In [17], replacing GPT-3.5 with GPT-4o reduced most error types, but not those with strong logic features (Section 4.2). Rumiantsau and Fokeev [36] compared three frontier models from two vendors (Claude Opus 4.7, Claude Sonnet 4.6 and GPT-5.4) on 100 retail analytics questions, each given the schema alone and the schema together with a hand-authored document of business semantics of about 4 KB. The document raised the pass rate by 17 to 23 percentage points for every model; within each condition, the three models were statistically indistinguishable, indicating that the presence of the document rather than the choice of model accounted for the significant differences. Several failure classes removed by the document correspond to categories of Section 3: questions on "sales" resolved against the wrong fact table and "last quarter" read relative to today rather than to the last date in the data are instances of Category K.3, a business term with a meaning that is not fixed by the schema. Sentinel keys and string-coded flags are instances of Category F, while metrics approximated instead of computed by their formula correspond to the derivations that the column catalog records for Category B and daily inventory snapshots summed across time are a grain error, close to Category E. Such a convergence despite an application domain (enterprise analytics) and method (paired benchmarking) different from ours is consistent with the view that the categories are properties of schema-grounded SQL generation rather than artifacts of our five-table teaching schema. Like the convergence with [17], this is not the same as validation; in [36], a single-shot protocol was used, so its evidence bears on the context pack, not on the execute–observe–correct loop.

A third study with a different design also sharpens the distinction between the two claims. Hasan [42] evaluated four open-weight and commercial LLMs, with between 7B and 70B parameters on 100 Spider schemas subjected to controlled schema changes (renamed columns and tables, changed types, dropped columns), then classified failures with a six-type taxonomy of structured hallucinations. For SQL output, the only type they checked was a reference to a table or column absent from the supplied schema, i.e., our Categories A and B. Two results bear on the present discussion. First, such references occurred for all four models even though every prompt contained the schema and instructed the model to use only its tables and columns; in a few cases, the invented identifier was the name that the column had before it was renamed, i.e., the name was recalled rather than being read. This is direct evidence for the cause assigned to Categories A and B in Section 4.3, namely, a conflict between parametric and contextual knowledge, and shows why the mitigation of Section 3 does not stop at an instruction but adds a check against the system catalog. Second, the four models shared most of the hallucination types, but in markedly different proportions; the largest model did not hallucinate less than the smaller ones, only differently. This is the pattern we stated at the beginning of this subsection, consisting of a common set of categories with model-dependent frequencies. From these differing profiles, it was concluded in [42] that mitigation should be model-specific. This conclusion concerned

postprocessing steps that target a model's dominant failure type. The context pack in our Section 3 instead addresses all fifteen categories at once, and its checks run regardless of the model profile, so applying it does not require knowing the profile. Whether each check is equally effective on every model is a question of frequency, however, and is not tested here. As in [36,42], single-shot generation was used in [42] without execution, so its evidence bears on generation, not on the execute–observe–correct loop.

A second claim that is not tested here (Section 1.3) is that frequency and severity are expected to vary with the model and with the quality of the context pack. In [36], it was noted that the effect of the document is bounded by its quality and coverage; the same holds for the seven files in our context pack. Therefore, our taxonomy should be read as a property of schema-grounded text-to-SQL generation that we expect to hold for other models, not as a finding verified on them. The baseline outputs are illustrative rather than elicited from a model (Section 2.5); measuring which of them current models actually produce and how often, across both models and context packs, is part of future work.

5.4. Model Tier and Size of the Context Pack

The use of a Sonnet-tier model is a consequence of the zero-cost requirement (Section 1), not a limitation accepted for convenience. It also has a pedagogical advantage: the failure modes catalogued here are those that students will actually meet with the model they can access free of charge. The findings of [36] add a further reason: with the business document in context, Claude Sonnet 4.6 was statistically indistinguishable from the larger Claude Opus 4.7, and each of the three models with the document outperformed each of them without it. Context, which the zero-cost configuration provides, mattered more than tier. A comparison across model sizes points the same way: in [42], the largest of four models did not produce fewer hallucinations than the smaller ones, only a different mix of them. A paid tier may reduce the frequency of some categories, but a mitigation strategy validated only on a paid model would leave the students who cannot afford it exposed to precisely the failures it claims to address. The countermeasures of Section 3 are designed so that their effectiveness does not depend on the tier: the same rules, files, and checks are available on any model that supports tool use.

Where the tier does make a difference is in two practical constraints, which are lesser concerns in an introductory course than they might be elsewhere. The first is usage: Claude Projects are available on the free plan, up to five per account [35], but the free plan has usage limits, and an agentic session consumes them faster than a plain chat because every execute–observe–correct cycle adds tool calls and their results to the conversation. Thus, a long laboratory session might be interrupted before a student has finished. Whether this happens in practice has not been measured. Detection methods that regenerate the query for several rewordings of the question, such as [31], would consume these limits faster still (about 7900 tokens per question with GPT-4o in [31]), while each check of Section 3 adds one or two statements. The second constraint is the size of the context pack. The Project instructions and Project knowledge are read in full in every conversation as long as they fit in the context window. On the free plan, this is a hard limit: the whole pack must fit or it cannot be used as a whole. On paid plans, once the Project knowledge approaches the context window, Claude switches automatically to retrieval-augmented generation (RAG) and reads only the passages it judges relevant to the question [35,43]. A DDL that is only partly retrieved reopens precisely the schema-level failures that the pack is meant to prevent (Categories A–C). In [36], a consistent observation was reported in passing: a system that reached the same business knowledge through tool-based search did not outperform the schema-only condition, whereas the whole document placed in the prompt

did. Therefore, a paid tier does not remove the size problem, only moving it from a capacity limit to a risk of partial retrieval.

A context pack of a few pages runs into neither of these constraints, and for a first course on relational database systems the pack is in fact modest. This is the experience of 27 years of teaching such a course, and it matches the size of the running examples used in the reference textbooks. Fundamentals of Database Systems by Elmasri and Navathe [39], first published in 1989 and now in its seventh edition, is among the most widely adopted database textbooks in computer science and engineering curricula worldwide, and has been translated into several languages. It introduces the basic SQL constructs (Chapters 6–7 of the seventh edition) on the COMPANY schema, which has just six relations (`employee`, `department`, `dept_locations`, `project`, `works_on`, `dependent`) populated with eight employees, three departments, and six projects. The running schema of the present paper (Section 2.1) is smaller still, with five tables. At this scale, the whole context pack is read in full on any plan, so neither the larger capacity of a paid tier nor the risk of partial retrieval comes into play.

Smaller does not necessarily mean poorer, however; what the taxonomy depends on is not the number of tables but the repertoire of query patterns that a schema can host, and on this score the two schemas are equivalent. Both contain a self-referencing foreign key (`super_ssn` in COMPANY, `manager_id` here), which Elmasri and Navathe themselves use to introduce recursive queries in SQL and on which Category L rests. Both resolve a many-to-many relationship through an associative table with attributes of its own (`works_on` with `hours`; `order_items` with `quantity` and `unit_price`), which supports multi-hop join paths (Category C), aggregation at two grains (Category E), and relational division. The textbook's classic request for the employees who work on all the projects of a department has its exact counterpart in the collective reading of Category K.1 (the products ordered by every customer). Both have a parent table with two independent one-to-many branches (`employee` with `works_on` and `dependent`; `employees` with their `orders` and their direct reports), the precondition of fan-out (Category D); a nullable foreign key that marks the top of the hierarchy (Category I); and date, numeric, and low-cardinality coded columns (Categories F, H, and J). None of the five tables is decorative; each carries at least one feature on which a worked example of Section 3 depends. The fifteen categories rest on the running schema; in turn, the running schema rests on the same constructs on which generations of students have learned SQL. Its compactness removes tables, not query patterns. The one structural feature of COMPANY that the running schema lacks is two distinct relationships between the same pair of tables, which would add a variant of Category C (a join along a declared but wrong foreign key); the category itself is unaffected.

5.5. Scope of the Proposal

The same constraint marks the boundary of the proposal anticipated in Section 2.2. This paper addresses classroom-size databases for which the context pack is read in full in every conversation. Industrial databases are outside its scope, for two reasons. First, their schemas are of a different order of magnitude: enterprise schemas have hundreds to thousands of columns, and the accuracy of LLM-based text-to-SQL falls sharply as schema size grows (from about 91% on Spider to 17–21% on Spider 2.0 and close to zero on BEAVER, a benchmark built from real data warehouses; [36], Section 2.3). Second, at that scale the pack could no longer be read in full, and the free plan would not suffice, so the zero-cost requirement could not be met. The categories themselves would not disappear: the schema-level ones (A–C) are likely to become more frequent as schemas grow. What does not transfer is the mitigation as designed here, consisting of a pack read in full on a free plan. The boundary follows from the purpose stated in Section 1.3: our proposal is a

teaching instrument for a first course on relational databases, not a text-to-SQL system for production.

5.6. Limitations

The limitations of the study, stated previously where they arise, are gathered here. Three elements are enforced outside the model: (a) PostgreSQL rejects non-existent relations, columns, and functions (A, B, G); (b) the read-only role is enforced by PostgreSQL (Section 2.5); and (c) every check is an executed query for which the text and result appear in the conversation, so compliance is visible to the user (the reason for the user actor in categories F, K, M, and N). The baseline outputs of Section 3 are constructed by the author rather than recorded from a model (Section 2.5). They show that a failure mode is plausible and what it would look like, not that a given model produces it or how often.

Preliminary checks run in November 2026 with three current assistants (Claude Sonnet 5, Gemini, ChatGPT), with one or two runs per request and with the request alone without the schema, indicate that the behavior is model- and run-dependent: Categories A, D, F, H, and K, for example, were all observed with at least one assistant, whereas B, C, G, I, J, and L were avoided by all of the tested assistants. When the fifteen requests were given to Claude in a single conversation together with the schema DDL, none of the failures was observed, and for the ambiguous requests of Category K the adopted readings were stated. These checks do not bear on the purpose of the taxonomy, which is not to predict what a given assistant returns in a given run (Section 1.3) but to give students a checklist of what to verify in any answer; they are consistent with the premise of the mitigation, namely, that context supplied from outside the model prevents most failures, the schema DDL being the first of the seven knowledge files. They are not reported as a measurement, only to clarify the conditional status of the baselines. All examples use a single five-table schema (Section 1.3).

Identification, merging, splitting, and the mapping onto prior taxonomies (Table 5, entries marked †) rest on the author's judgment, with no inter-rater agreement. The mitigations were checked on the worked example of each category, not measured over a set of requests, and the MCP server used in the mitigated configuration has since been deprecated (Section 2.5). Except for the errors that PostgreSQL reports and the read-only role, the checks of Section 3 are run only if the model follows the standing rules; an external validator that parsed each candidate query before execution, checking its table and column references against the system catalog, its join predicates against the declared foreign keys, and its aggregations for one-to-many branches joined before grouping, would turn several of these checks into guarantees, but requires programming and a component outside the configuration of Section 2.3, where the setup involves none. Finally, the zero-cost configuration depends on features of a commercial product, such as the availability of Projects on the free plan and its usage limits, which the vendor may change.

In sum, the effectiveness of the mitigation strategy and the generalizability of the approach to other schemas, models, and student cohorts have not yet been systematically validated.

6. Conclusions

As a conceptual and illustrative contribution, this paper proposes a fifteen-category taxonomy of the hallucinations that an agentic LLM can produce in text-to-SQL generation, designed as a teaching instrument for a first course on relational databases. The categories were identified through an expert-driven process and illustrated against a five-table PostgreSQL schema, each by a constructed baseline output and by a mitigated configuration run with Claude Sonnet 5. Comparison with prior work (Section 4) shows that nine categories

have a direct, close, or partial counterpart in existing text-to-SQL taxonomies; two of these (fan-out and recursive traversal) have no dedicated counterparts, although their effects are observable as wrong results, while four (silent resolution of ambiguity, explanation–query inconsistency, fabricated execution, and false security assurances) cannot be represented in a static evaluation because they concern the answer returned to the user rather than the SQL query.

Our proposal does not ask students to learn fifteen countermeasures, only that the instructor prepare a single reusable context pack once per course, much as an instructor today prepares the laboratory database, then show students how to store it in a Claude Project of their own and connect it to the course database. The pack is made up of the standing rules and the seven reference files. Every subsequent conversation opened within the Project starts with the whole pack in context, whether the instructor is demonstrating in class or a student is working in the laboratory or at home. As designed, and as illustrated on our worked examples, most of the mitigation takes place without the user's intervention: the engine rejects non-existent tables, columns, and functions, while the loop runs the checks that the standing rules prescribe. Two qualifications apply. First, the checks run only when a database connection is configured, while the model follows the rules; without a connection, the rules make the assistant present each query as not executed (Section 2.4). Second, four of the categories leave part of the work with the user: deciding the meaning of a business term that the glossary does not define, choosing among the readings of an ambiguous request, comparing an explanation with the executed query and its rows, and checking that every reported number comes with the query that produced it. These are habits that the course aims to leave with students for times when no context pack is available.

The claims of this paper are bounded accordingly. The categories are expected to be model-agnostic, as properties of schema-grounded SQL generation, and three independent studies are consistent with this reading (Section 5.3); their frequency and severity and the effectiveness of the countermeasures are not measured here, and the generalizability of our approach has not yet been systematically validated (Section 5.6). Our proposal addresses classroom-size databases where the context pack can be read in full on the free plan; industrial databases are outside its scope (Section 5.5). Finally, the taxonomy is a working one, not necessarily closed; extending it to the constructs excluded in Section 2.5, such as window functions, materialized views, multi-statement transactions, and data modification statements is likely to bring further failure modes to light.

Future Work

The taxonomy and its mitigations are illustrated here through one worked example per category, with a constructed baseline and a mitigated configuration run with Claude Sonnet 5; they have not yet been validated with a student cohort or measured over a set of requests. Four lines of future work follow from these boundaries.

Classroom deployment. We plan to deploy the context pack and the execute–observe–correct loop in our database course and compare the learning outcomes of students working with and without the mitigations. The deployment will also record how often each category occurs in students' sessions. It will use the exercise suggested in Section 5.1, in which the same request is put to an assistant without and with the context pack, after which students compare the answers they actually obtain with each other and with the illustrative baseline of Section 3. Since agentic sessions with repeated tool calls consume the usage limits of the free plan faster than plain chat (Section 5.4), the deployment will assess whether those limits are compatible with a full laboratory session and how institutional licenses compare

with the zero-cost configuration in terms of equity of access. A maintained MCP server will be used for PostgreSQL, combined with a read-only database role (Section 2.5).

Sampling-based characterization. For each category, a follow-up study should quantify how often the hallucination recurs across repeated generations of the same request, how sensitive it is to rewordings of the request, and how the taxonomy behaves under adversarial or deliberately under-specified prompting. Repetitions and rewordings are not interchangeable: in [31], it was reported that resampling the same question tends to return nearly identical SQL, whereas rewording it exposes inconsistencies that resampling hides, so the study will rely mainly on rewordings. The reference-free detector of [31] provides a baseline against which the hand-written checks of the context pack can be compared on the same requests. The study should extend to other agentic LLMs and follow the profile-oriented evaluation of [42], which reports the distribution of hallucination types for each model rather than a single error rate.

Quality of the context pack. The effect of a context pack is bounded by its quality and coverage [36]. An ablation study in which one knowledge file or one standing rule at a time is removed or degraded would show which component prevents which category. The same study should address schema evolution, which was shown in [42] to be a source of hallucination in its own right: when the instructor releases a new version of the course database, the pack of the previous version describes a database that no longer exists, and only the run-time checks against the system catalog and data can reveal the mismatch.

From prescribed checks to guarantees. Except for the errors that PostgreSQL reports and the read-only role, the checks of Section 3 run only if the model follows the standing rules (Section 5.6). An external validator that parses each candidate query before execution while checking its references against the system catalog, its join predicates against the declared foreign keys, and its aggregations for one-to-many branches joined before grouping would turn several of these checks into guarantees. Building this would requires programming, which the configuration proposed here deliberately avoids; whether it can be offered to students as a ready-made component compatible with the zero-cost requirement remains an open question.

Supplementary Materials: The following supporting information can be downloaded at https://www.mdpi.com/article/10.3390/app162311607/s1.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data supporting the reported results are provided as Supplementary File S1, a zip archive containing: (a) the course database, as a PostgreSQL 17 script with the DDL and the data of the five tables; (b) the standing rules, as entered in the instructions of the Claude Project (the text of Table A1); and (c) the seven files of the Project knowledge (Table A2). The configuration of the PostgreSQL MCP server and the transcripts of the mitigated configuration sessions are available from the corresponding author on request.

Acknowledgments: During the preparation of this manuscript, the author used ChatGPT 6, Claude, and Gemini 4 Hydrogen for the purposes of grammar and spelling check. Moreover, all figures were constructed through iterative interaction with Claude (Anthropic, Claude Sonnet 5). The author has reviewed and edited the output and takes full responsibility for the content of this study.

Conflicts of Interest: The author declares no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

E-R	Entity Relationship
DBMS	Database Management System
DB	Database
DDL	Data Definition Language
PK	Primary Key
FK	Foreign Key
SQL	Structured Query Language
CTE	Common Table Expression
NL	Natural Language
LLM	Large Language Model
NL2SQL	Natural Language-to-SQL
ICL	In-Context Learning
MCP	Model Context Protocol
RAG	Retrieval-Augmented Generation

Appendix A. Implementation in a Claude Project

In practice, the instructor does not distribute fifteen separate additions, one per category; since a user cannot know in advance which category a question will fall into, all of them are merged into a single context pack, which every user stores once in a Claude Project: the standing rules, entered in the Project instructions, and the seven files of the Project knowledge described in Section 2.2. The pack is part of C_0 in every session (Section 2.4), and the context needed to prevent or mitigate each failure mode is present from the first question onward rather than being added after an error has occurred. Tables A1 and A2 state what the pack contains and in what words: Table A1 gives the standing rules as entered in the Project instructions, while Table A2 provides the content of the seven files of the Project knowledge.

Table A1. Standing rules as entered in the instructions of the Claude Project: a general rule on loop control, followed by one rule per hallucination category. File names refer to the Project knowledge (Section 2.2). Rules marked (∗) apply to every query and open the instructions.

Cat.	Standing Rule (Project Instructions)
All	(∗) Loop control. Present a query as verified only if it was executed and passed every check that these rules prescribe for it. If a check still fails after three revisions of the query, stop revising, present the last version as not verified, and name the check that failed.
A	(∗) Use only the tables declared in `01_schema.sql`. If a request needs data that no declared table holds, say so and do not write a query for it. Before presenting any query, execute it with the PostgreSQL MCP tool; if the engine reports that a relation does not exist, revise the query and execute it again. If no database tool is available in this conversation, present the query as not executed.
B	Reference only columns listed as STORED for that table in `02_column_catalog.txt`. If a requested value is not STORED (e.g., a total, a revenue, a margin), compute it with the derivation given in the catalog and say so in a comment. Before presenting, verify every (table, column) pair against `information_schema.columns`.
C	Join two tables only along the edges listed in `03_fk_graph.txt`, directly or through a chain of listed edges; when no direct edge exists, use the mandatory path given there. A shared data type or a similar column name is not evidence of a relationship. Before executing, check the column pair of every join predicate against the foreign keys declared in `pg_catalog.pg_constraint`, and revise any join whose pair is not a declared edge.
D	Grain safety. When a query combines two or more one-to-many relationships that start from the same parent table, aggregate each relationship to the parent's grain in its own CTE before joining the results. State the grain of the final result in a comment (e.g., "one row per `employee_id`"). Before presenting, verify that the combined result (without an outer `GROUP BY`) has `COUNT(*) = COUNT(DISTINCT parent key)`, and that each metric equals the same metric computed from its own branch alone.

Table A1. *Cont.*

Cat.	Standing Rule (Project Instructions)
E	Aggregation scope. When a request asks for an aggregate per X of a metric that is itself the result of an aggregation (e.g., average ORDER value per country, where order value is the sum of an order's lines), first compute the metric at its own grain in a CTE, as defined in `02_column_catalog.txt` and `05_business_glossary.txt`; state that grain in a comment; and only then aggregate per X. Before presenting, verify that, in every group, the number of values entering the outer aggregate equals the number of distinct keys at the metric's grain (e.g., `COUNT(DISTINCT order_id)` per country).
F	Before filtering an enumerated column, run SELECT DISTINCT on it to confirm its current values (`04_value_catalog.txt` may be out of date). If the user's word differs from a stored value only in spelling or case (e.g., "canceled" vs "cancelled"), use the stored value and say so. If it differs in meaning (e.g., "completed"), apply the definition in `05_business_glossary.txt`; if the term is not defined there, ask the user. Never filter on a value that the probe did not return.
G	(∗) Target dialect: PostgreSQL 17 only. Do not use functions or syntax of other DBMSs. If the engine reports that a function does not exist, rewrite the query with the PostgreSQL equivalent and execute it again.
H	Before dividing, make sure at least one operand is `NUMERIC`: division between integers truncates. Flag for the user any predicate that compares columns of different declared types.
I	Test `NULL` only with `IS NULL/IS NOT NULL`, never with = or <>, following the pattern in `06_exemplars.sql`. For any filter on a nullable column, compare the row count of your predicate with the row count of its `IS NULL/IS NOT NULL` counterpart and explain any difference before presenting. For a predicate with = or <> on a nullable column, also count the rows where the column `IS NULL`, and state whether they are meant to be included.
J	Temporal ranges. Express every date or time range as a half-open interval, col >= start AND col < start of the next period (e.g., `order_date >= DATE '2024-02-01'` AND `order_date < DATE '2024-03-01'`), whatever the column type; never use `BETWEEN`, and never compute the last day of a period by hand. Take the type of the column (`DATE` or `TIMESTAMP`) from `01_schema.sql` and follow the pattern in `06_exemplars.sql`. Before presenting, count the selected rows a second time with a predicate that does not depend on the length of the period (`date_trunc` on the column) and check that the two counts coincide. When `07_reference_queries.sql` lists a boundary query for the period, run it and compare its result with the expected one recorded there.
K	K.1 Scope: treat "each", "every", "all" applied to a group as ambiguous between the distributive and the collective reading. K.2 Attachment: when a modifier follows two entities ("customers and orders that are cancelled"), check whether it applies to one or both. K.3 Vagueness: resolve business terms with `05_business_glossary.txt`; a term not in the glossary requires a clarification. In all three cases, never resolve silently: name the ambiguity, state the reading chosen, and either ask the user or present the alternative readings side by side, each labelled.
L	A self-referencing foreign key (flagged in `03_fk_graph.txt`) signals a hierarchy. For requests over "all levels", "indirect", "whole chain" or similar, write a `WITH RECURSIVE` query following the pattern in `06_exemplars.sql`, never a single self-join. Before presenting, run it on the reference sub-hierarchy and compare the result with the expected answer in `07_reference_queries.sql`.
M	Execute the query before explaining it. Derive the explanation from the SQL text actually executed, clause by clause, and from the rows it returned, never from the intended logic. Before presenting, compare the row count with an independent count of the same set (e.g., for products never ordered: the number of products minus the number of distinct products in `order_items`). When a reference query for the same request exists in `07_reference_queries.sql`, compare the two results before presenting the query or its explanation.
N	(∗) Never state a number, a row count or a row value that was not returned by a query executed with the MCP tool in this conversation. Show the executed query next to every reported result. If a query was not executed, say so explicitly and do not give a result.
O	For any query template that will receive user input, use parameterized statements only (e.g., `WHERE customer_name` = $1); never concatenate user input into SQL text; assume a least-privilege, read-only database role. Check every template against these three points before presenting it, whatever your own assessment of its safety; read-only access does not prevent injection. Test the template by executing it with a legitimate value that contains an apostrophe (e.g., O'Brien), following the pattern in `06_exemplars.sql`.

Table A2. Reference material uploaded as files to the knowledge of the Claude Project, per hallucination category. "–" = no file needed, that is, the rule in Table A1 is self-contained. Files serving several categories are listed in each row that uses them.

Cat.	Project Knowledge (File and Content)
A	`01_schema.sql`: the DDL of the five tables of the course database (`CREATE TABLE` statements with primary and foreign keys), the same DDL as in the baseline; the only file that declares which tables exist.
B	`02_column_catalog.txt`: for every attribute of the five tables, (a) whether it is STORED or must be DERIVED, with the derivation of each derived attribute (e.g., `order value = SUM(quantity × unit_price)` over `order_items`, per `order_id`); (b) a one-line business meaning, where the name does not make it obvious.
C	`03_fk_graph.txt`: the five direct edges between tables, the mandatory path for every pair of tables without a direct edge, and the statement that no other relationship exists.
D	`03_fk_graph.txt`: for each parent table, its one-to-many relationships (`employees` → `orders`; `employees` → `employees`, as manager; `orders` → `order_items`; `customers` → `orders`; `products` → `order_items`), so that a parent with two "many" branches, here employees, is visible at a glance.
E	`02_column_catalog.txt`: the derivation of order value and the grain of each metric (order value at the order grain, line value at the order-line grain). `05_business_glossary.txt`: "average order value" defined as the mean, over orders, of the order value.
F	`04_value_catalog.txt`: the stored values of every low-cardinality column, with the date of the snapshot. `05_business_glossary.txt:` a completed order defined as an order with status 'delivered'.
G	–
H	–
I	`06_exemplars.sql`: a `NULL` test written with `IS NULL`.
J	`01_schema.sql`: the declared type (`DATE` or `TIMESTAMP`) of each date or time column. `06_exemplars.sql`: a monthly range written as a half-open interval. `07_reference_queries.sql`: a boundary query for February 2024, with its expected result.
K	`06_exemplars.sql` (K.1): a labelled pair of queries for a request with a quantifier, a distributive one (one row set per member of the group) and a collective one (relational division: `GROUP BY ... HAVING COUNT(DISTINCT ...)` equal to the size of the group). `06_exemplars.sql` (K.2): the narrow reading of an attachment ambiguity, an outer join with the filter on the inner table placed in the `ON` clause rather than in `WHERE`. `05_business_glossary.txt` (K.3): "top customers" defined as the customers ranked by revenue, i.e., `SUM(quantity × unit_price)` over the lines of their non-cancelled orders, with a default list length of 10, ties at the cut-off included.
L	`03_fk_graph.txt`: `employees.manager_id` → `employees.employee_id` flagged as a self-referencing edge, i.e., as a hierarchy. `06_exemplars.sql`: a `WITH RECURSIVE` query on employees that returns every pair (manager, direct or indirect report) with its depth, and guards against cycles. `07_reference_queries.sql`: for one manager of a reference sub-hierarchy, the expected set of direct and indirect reports, with their depth.
M	`07_reference_queries.sql`: for the request on products never ordered, the anti-join written with `NOT EXISTS`, with its expected result.
N	–
O	`06_exemplars.sql`: a parameterized template, both in SQL (`PREPARE ... EXECUTE`) and in the placeholder form of the host language used in the course (e.g., %s with psycopg in Python).

References

1. Shi, Y.; Yu, K.; Dong, Y.; Chen, F. Large language models in education: A systematic review of empirical applications, benefits, and challenges. *Comput. Educ. Artif. Intell.* **2026**, *10*, 100529. https://doi.org/10.1016/j.caeai.2025.100529.
2. Sun, R.; Deng, X. Using Generative AI to Enhance Experiential Learning: An Exploratory Study of ChatGPT Use by University Students. *J. Inf. Syst. Educ.* **2025**, *36*, 53–64. https://doi.org/10.62273/ZLUM4022.
3. Gimpel, H.; Hall, K.; Decker, S.; Eymann, T.; Gutheil, N.; Lämmermann, L.; Braig, N.; Maedche, A.; Ruiner, C.; Vandirk, S.; et al. Using Generative AI in Higher Education: A Guide for Instructors. *J. Inf. Syst. Educ.* **2025**, *36*, 237–256. https://doi.org/10.62273/QLLG7172.

4. Chen, D.-L.; Balasubramanian, P.; Loven, L.; Pirttikangas, S.; Sauvola, J.; Kostakos, P. Integrating Generative AI-enhanced Cognitive Systems in Higher Education: From Stakeholder Perceptions to a Conceptual Framework Considering the EU AI Act. *arXiv* **2026**, arXiv:2602.10802.
5. Peláez-Sánchez, I.C.; Velarde-Camaqui, D.; Glasserman-Morales, L.D. The impact of large language models on higher education: Exploring the connection between AI and Education 4.0. *Front. Educ.* **2024**, *9*, 1392091. https://doi.org/10.3389/feduc.2024.1392091.
6. Dai, K.; Liu, Y.; Zhang, X. Generative AI in higher education: A bibliometric review of emerging trends, power dynamics, and global research landscapes, *Comput. Educ. Artif. Intell.* **2026**, *10*, 100544. https://doi.org/10.1016/j.caeai.2026.100544.
7. Yigci, D.; Eryilmaz, M.; Yetisen, A.K.; Tasoglu, S; Ozcan, A. Large Language Model-Based Chatbots in Higher Education. *Adv. Intell. Syst.* **2025**, *7*, 2400429. https://doi.org/10.1002/aisy.202400429.
8. Miedema, D.; Bhowmick, S.S.; Liut, M. Third International Workshop on Data Systems Education (DataEd'24). In Proceedings of the Companion of the 2024 International Conference on Management of Data (SIGMOD Companion'24), Santiago, Chile, 9–15 June 2024. https://doi.org/10.1145/3626246.3655018.
9. Riazi, S.; Rooshenas, P. LLM-Driven Feedback for Enhancing Conceptual Design Learning in Database Systems Courses. In Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025), Pittsburgh, PA, USA, 26 February–1 March 2025. https://doi.org/10.1145/3641554.3701940.
10. Zhan, Y.; Chen, L. Teaching Case: Harness the Power of Interactive Large Language Model in Teaching Using a Capstone Project in the Database Management Course. *J. Inf. Syst. Educ.* **2025**, *36*, 224–236. https://doi.org/10.62273/VMJS9132.
11. Utkan, K.T.; Öztayşi, B. Large Language Models for Automated Personalized Feedback in SQL Education: A Preliminary Study. In *International Conference on Intelligent and Fuzzy Systems*; Kahraman, C., Cebi, S., Oztaysi, B., Onar, S.C., Tolga, C., Sari, I.U., Otay, I., Eds.; Springer: Cham, Switzerland, 2025; Volume 1529. https://doi.org/10.1007/978-3-031-97992-7_70.
12. Pornphol, P.; Chittayasothorn, S. Using LLM Artificial Intelligence Systems as Complex SQL Programming Assistants. In Proceedings of the 12th International Conference on Information and Education Technology (ICIET), Yamaguchi, Japan, 18–20 March 2024; pp. 477–481. https://doi.org/10.1109/ICIET60671.2024.10542806.
13. Moser, S.; Pustulka, E. Enhancing SQL Learning with Large Language Models: Generating Translations and Explanations for Educational Use. In *Proceedings of the 4th International Conference on Innovations in Computing Research (ICR'25)*; Daimi, K., Alsadoon, A., Eds.; Springer: Cham, Switzerland, 2025; Volume 1487. https://doi.org/10.1007/978-3-031-95652-2_23.
14. Wanjiru, B.; van Bommel, P.; Hiemstra, D. Sensitivity of Automated SQL Grading in Computer Science Courses. In *Proceedings of the the 4th International Conference on Innovations in Computing Research (ICR'24)*; Daimi, K., Al Sadoon, A., Eds.; Springer: Cham, Switzerland, 2024; Volume 1058. https://doi.org/10.1007/978-3-031-65522-7_26.
15. Zhang, X. Teaching Tip: Incorporating AI Tools Into Database Classes. *J. Inf. Syst. Educ.* **2025**, *36*, 37–52. https://doi.org/10.62273/GKZI2477.
16. Qu, G.; Li, J.; Li, B.; Qin, B.; Huo, N.; Ma, C.; Cheng, R. Before Generation, Align it! A Novel and Effective Strategy for Mitigating Hallucinations in Text-to-SQL Generation. In Proceedings of the Findings of the Association for Computational Linguistics: ACL 2024, Bangkok, Thailand, 11–16 August 2024; pp. 5456–5471. https://aclanthology.org/2024.findings-acl.324/.
17. Shen, J.; Wan, C.; Qiao, R.; Zou, J.; Xu, H.; Shao, Y.; Zhang, Y.; Miao, W.; Pu, G. Understanding, Detecting, and Repairing Real-World In-Context-Learning-Based Text-to-SQL Errors. *Proc. ACM Softw. Eng.* **2025**, *3*, 3722–3745. https://doi.org/10.1145/3808171.
18. Hoang, M.; Madisetti, V.K. To RAG or Not to RAG, That Is the Question: Effective Text-to-SQL Generation Under Ambiguity. *IEEE Access* **2026**, *14*, 35440–35458. https://doi.org/10.1109/ACCESS.2026.3670064.
19. Ponzini, D.; Catania, B.; Guerrini, G. Enhancing SQL Learning Through Generative AI and Student Error Analysis. In *New Trends in Database and Information Systems (ADBIS 2025). Communications in Computer and Information Science*; Chrysanthis, P.K., Nørvåg, K., Stefanidis, K., Zhang, Z., Quintarelli, E., Zumpano, E., Eds.; Springer: Cham, Switzerland, 2026; Volume 2676, pp. 118–128. https://doi.org/10.1007/978-3-032-05727-3_12.
20. Taipalus, T.; Siponen, M.; Vartiainen, T. Errors and Complications in SQL Query Formulation. *ACM Trans. Comput. Educ.* **2018**, *18*, 15. https://doi.org/10.1145/3231712.
21. Miedema, D.; Aivaloglou, E.; Fletcher, G. Identifying SQL Misconceptions of Novices: Findings from a Think-Aloud Study. In *Proceedings of the 17th ACM Conference on International Computing Education Research (ICER 2021)*; ACM: New York, NY, USA, 2021; pp. 355–367. https://doi.org/10.1145/3446871.3469759.
22. Miedema, D.; Fletcher, G.; Aivaloglou, E. Expert Perspectives on Student Errors in SQL. *ACM Trans. Comput. Educ.* **2022**, *23*, 11. https://doi.org/10.1145/3551392.
23. Balderas, A.; Baena-Pérez, R.; Person, T.; Mota, J.M.; Ruiz-Rube, I. Chatbot-Based Learning Platform for SQL Training. *Int. J. Eof Interact. Multimed. Artif. Intell.* **2024**, *8*, 135–145. https://doi.org/10.9781/ijimai.2022.05.003.

24. AlRabah, A.; Yang, S.; Alawini, A. Optimizing Database Query Learning: A Generative AI Approach for Semantic Error Feedback. In Proceedings of the ASEE Annual Conference and Exposition, Conference Proceedings, American Society for Engineering Education, Portland, OR, USA, 23–26 June 2024. Available online: https://api.semanticscholar.org/CorpusID:271713370 (accessed on 8 October 2026).
25. Cagliero, L.; Farinetti, L.; Fior, J.; Manenti, A.I. ChatGPT, be my Teaching Assistant! Automatic Correction of SQL Exercises. In Proceedings of the IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC), Osaka, Japan, 2–4 July 2024; pp. 81–87. https://doi.org/10.1109/COMPSAC61105.2024.00021.
26. Kasneci, E.; Seßler, K.; Küchemann, S.; Bannert, M.; Dementieva, D.; Fischer, F.; Gasser, U.; Groh, G.; Günnemann, S.; Kasneci, G.; et al. ChatGPT for Good? On Opportunities and Challenges of Large Language Models for Education. *Learn. Individ. Differ.* **2023**, *103*, 102274. https://doi.org/10.1016/j.lindif.2023.102274.
27. Lee, A.; Chen, K. Arguing with AI: LLMs as a Debate Partner in the Classroom. In Proceedings of the IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE), Macao, China, 4–7 December 2025; pp. 1–5. Available online: https://api.semanticscholar.org/CorpusID:284969174 (accessed on 8 October 2026).
28. Bernstein, S.; Rahman, A.; Sharifi, N.; Terbish, A.; MacNeil, S. Beyond the Benefits: A Systematic Review of the Harms and Consequences of Generative AI in Computing Education. In Proceedings of the 25th Koli Calling International Conference on Computing Education Research (Koli Calling'25), Koli, Finland, 11–16 November 2025. https://doi.org/10.1145/3769994.3770036.
29. Park, J.; Lim, K.; Park, S.; Kim, D. Supporting Students in Navigating LLM-Generated Insecure Code. *arXiv* **2025**, arXiv:2511.20878.
30. López-Fernández, D.; Vergaz, R. ChatGPT in Computer Science Education: A Case Study on a Database Administration Course. *Appl. Sci.* **2025**, *15*, 985. https://doi.org/10.3390/app15020985.
31. Lin, L.; Zhu, Q.; Hong, J.; Wang, C.; Liu, Y.; Wu, R. Validating LLM-Generated SQL Queries through Metamorphic Prompting. *Proc. ACM Softw. Eng.* **2026**, *3*, 411–434. https://doi.org/10.1145/3797146.
32. Liu, X.; Shen, S.; Li, B.; Tang, N.; Luo, Y. NL2SQLBUGs: A Benchmark for Detecting Semantic Errors in NL2SQL Translation. In Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining Vol.2 (KDD '25), Toronto, ON, Canada, 3–7 August 2025; pp. 5662– 5673. https://doi.org/10.1145/3711896.3737427.
33. Saparina, I.; Lapata, M. AMBROSIA: A benchmark for parsing ambiguous questions into database queries. *Adv. Neural Inf. Process. Syst.* **2024**, *37*, 90600–90628. https://doi.org/10.52202/079017-2876.
34. Huang, L.; Yu, W.; Ma, W.; Zhong, W.; Feng, Z.; Wang, H.; Chen, Q.; Peng, W.; Feng, X.; Qin, B.; et al. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Trans. Inf. Syst.* **2025**, *43*, 1–55. https://doi.org/10.1145/3703155.
35. Anthropic. What Are Projects? Available online: https://support.claude.com/en/articles/9517075-what-are-projects (accessed on 1 October 2026).
36. Rumiantsau, M.; Fokeev, I. Semantic Layers for Reliable LLM-Powered Data Analytics: A Paired Benchmark of Accuracy and Hallucination Across Three Frontier Models. *arXiv* **2026**, arXiv:2604.25149.
37. Anthropic. Plans & Pricing. Available online: https://claude.com/pricing (accessed on 8 October 2026).
38. UNESCO. *Recommendation on the Ethics of Artificial Intelligence Adopted by the General Conference on 23 November 2021*; UNESCO: Paris, France, 2022. Available online: https://unesdoc.unesco.org/ark:/48223/pf0000381137 (accessed on 1 October 2026).
39. Elmasri, R.; Navathe, S.B. *Fundamentals of Database Systems*, 7th ed.; Pearson: Hoboken, NJ, USA, 2016.
40. Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; Cao, Y. React: Synergizing reasoning and acting in language models. In Proceedings of the 11th International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, 1–5 May 2023.
41. Farinetti, L; Cagliero, L. A Critical Approach to ChatGPT: An Experience in SQL Learning. In Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025), Pittsburgh, PA, USA, 26 February–1 March 2025. https://doi.org/10.1145/3641554.3701932.
42. Hasan, M. StructHallu-Drift: Benchmarking Structured Hallucinations Under Schema Evolution in LLMs. In *Proceedings of the First Workshop on Structured Understanding, Retrieval, and Generation in the LLM Era (SURGeLLM 2026)*; Gupta, V., Ding, K., Kokel, H., Kokel, H., Zhao, Y., Agarwal, A., Wang, Y., Glass, M., Zhang, Y., Srinivas, K., Chen, X., et al., Eds.; Association for Computational Linguistics: Stroudsburg, PA, USA, 2026; pp. 333–343. https://doi.org/10.18653/v1/2026.surgellm-1.22.
43. Anthropic. Retrieval Augmented Generation (RAG) for Projects. Available online: https://support.claude.com/en/articles/11473015-retrieval-augmented-generation-rag-for-projects (accessed on 1 October 2026).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.