TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Kristo Isberg 214012IVSM

Detecting SQL Antipatterns in jOOQ Database Access Code Using Large Language Models

Master's Thesis

Supervisor: Erki Eessaar
PhD

Tallinn 2026

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Kristo Isberg 214012IVSM

SQL antimustrite tuvastamine jOOQ andmebaasikoodis suurte keelemudelite abil

Magistritöö

Juhendaja: Erki Eessaar

PhD

Tallinn 2026

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis and that this thesis has not been presented for examination or submitted for defense anywhere else. All used materials, references to the literature, and work of others have been cited.

Author: Kristo Isberg

11.05.2026

Abstract

Traditional static analysis tools struggle to detect SQL antipatterns in dynamically generated queries built with domain-specific languages like the Java library jOOQ. To address this limitation, this thesis develops an LLM-powered static analysis tool capable of identifying SQL antipatterns directly from jOOQ database access code.

The methodology involved creating a human-annotated dataset of 1562 antipattern occurrences from 61 open-source projects. An evaluation of four contemporary LLMs and various prompting strategies revealed that the Claude Opus 4.5 model using a Zero-Shot prompt achieved near-optimal detection performance at a low runtime, whereas complex reasoning prompts proved costly.

Deploying the finalised tool across 602 projects flagged 15 931 occurrences of seven distinct antipatterns. Results indicated that antipatterns frequently co-occur, with "Implicit Columns" and "ID Required" present in nearly 90 % of the projects. Furthermore, the analysis revealed that the vast majority of the detected query antipattern occurrences were associated with a few specific jOOQ convenience methods, such as `selectFrom` and `Field.like`. Ultimately, this research demonstrates the potential of LLMs to address the parsing limitations of dynamic database frameworks, offering actionable insights for improving software quality and API usage patterns.

The finalised tool is available in the following GitHub repository: https://github.com/kristoisberg/jooq-antipattern-detector. All other artefacts produced as parts of the thesis are available in the following GitHub repository: https://github.com/kristoisberg/masters-thesis.

The thesis is in English and contains 85 pages of text, 9 chapters, 48 figures, 73 tables.

Annotatsioon

SQL antimustrite tuvastamine jOOQ andmebaasikoodis suurte keelemudelite abil

Andmebaasipäringute optimeerimine ja andmetervikluse tagamine on tarkvaraarenduses kriitilise tähtsusega. Tarkvaraprojektides esinevad sageli korduvad disaini- ja päringuvead, mida nimetatakse SQL antimustriteks. Need ebaoptimaalsed lahendused halvendavad märkimisväärselt süsteemide jõudlust, hooldatavust ja töökindlust. Kuigi traditsiooniliste, staatiliste SQL-lausete kontrollimiseks on loodud mitmeid analüüsitööriistu, puudub praeguseni piisav tugi dünaamiliselt genereeritavatele lausetele. Traditsiooniline staatiline koodianalüüs ei suuda tõhusalt analüüsida koodi, mis kasutab andmebaasiga suhtlemiseks valdkonnapõhiseid keeli (DSL), näiteks populaarset Java teeki jOOQ.

Käesoleva lõputöö peamine eesmärk on arendada välja suurte keelemudelite (LLM) baasil töötav staatilise analüüsi tööriist, mis suudab tuvastada SQL antimustreid Java lähtekoodist, kus kasutatakse jOOQ teeki. Lisaks on töö teiseseks eesmärgiks analüüsida nende antimustrite levikut ja koosesinemist avatud lähtekoodiga tarkvaraprojektides ning selgitada välja, milliste jOOQ rakendusliidese (API) meetoditega antimustrite esinemisjuhud kõige sagedamini seostuvad.

Lõputöö metoodika ja teostus jagunesid neljaks põhietapiks. Esimeses etapis viidi läbi andmekaeve GitHubi keskkonnas, kust koguti üle 600 avatud lähtekoodiga Java projekti. Valimist eraldati 61 esinduslikku projekti, millest otsiti ja märgiti käsitsi üles 1562 SQL antimustri esinemisjuhtu. Loodud andmestiku toel hinnati teises etapis nelja 2026. aasta alguse seisuga kaasaegse suure keelemudeli (OpenAI GPT-5.2, Z.ai GLM-5, Anthropic Claude Opus 4.5 ja OpenAI gpt-oss-120B) ning nelja erineva viipamisstrateegia (Zero-Shot, Few-Shot, Chain-of-Thought, Tree-of-Thought) suutlikkust probleeme tuvastada.

Tulemused näitasid, et suured keelemudelid suudavad dünaamiliselt genereeritud andmebaasikoodist antimustreid leida suure täpsusega. Üllatuslikult ei parandanud keerukamad ja

arutlemist nõudvad viibad (Chain-of-Thought, Tree-of-Thought) järjekindlalt tuvastamise täpsust, kuid suurendasid märkimisväärselt analüüsi kulusid ja käitusaega. Seetõttu valiti lõpliku tuvastustööriista aluseks Zero-Shot viipamisstrateegia ning Claude Opus 4.5 mudel (ilma täiendava arutlusprotsessita), mis saavutas mitmeklassilise ja mitme esinemisjuhuga asukoha tuvastamise ülesandes kõrge F1-skoori (vähemalt 0,88).

Kolmandas etapis arendati välja iseseisev käsureatööriist, mis loeb sisendiks Java lähtekoodi ning suhtleb keelemudeliga antimustrite esinemisjuhtude tuvastamiseks. Neljandas etapis rakendati loodud tööriista algselt kogutud 602 projekti peal, et uurida antimustrite esinemissagedust. Empiiriline analüüs märgistas nendes projektides seitsme antimustri kohta kokku 15 931 esinemisjuhtu. Tulemused kinnitasid, et antimustrid on avatud lähtekoodiga projektides laialt levinud. Ülekaalukalt kõige sagedasemad antimustrid olid "Ilmutamata veerud" (*Implicit Columns*) ja "ID, palun" (*ID Required*), mis eksisteerisid vastavalt 89 % ja 87 % analüüsitud projektides.

Lisaks näitas analüüs sagedast koosesinemist spetsiifiliste jOOQ abstraktsioonimeetodite ja antimustrite esinemisjuhtude vahel. Lihtsustatud meetodite, nagu `selectFrom` ja `Field.like` (koos metamärkidega), kasutamine esines sageli samades koodilõikudes vastavate antimustritega. See viitab võimalikule korrelatsioonile nende meetodite kasutuse ja teatud vigade sageduse vahel.

Lõputöö demonstreerib suurte keelemudelite võimet analüüsida dünaamilisi andmebaasiraamistikke kasutavate projektide valdkonnapõhisel keelel põhinevat lähtekoodi. Lõputöö pakub väärtuslikku teavet nii tarkvaraarendajatele koodikvaliteedi parandamiseks kui ka teekide hooldajatele dokumentatsiooni ja võimalike rakendusliidese-disaini muudatuste kaalumiseks.

Lõputöö raames välja arendatud tööriist on saadaval GitHubi hoidlas: https://github.com/kristoisberg/jooq-antipattern-detector. Kõik muud lõputöö raames valminud artefaktid on saadaval GitHubi hoidlas: https://github.com/kristoisberg/masters-thesis.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 85 leheküljel, 9 peatükki, 48 joonist, 73 tabelit.

List of abbreviations and terms

AAII	Artificial Analysis Intelligence Index. An index used to benchmark and compare the performance and capabilities of different artificial intelligence models.
API	Application Programming Interface. A software intermediary that allows different applications or services to communicate and exchange data with one another.
AST	Abstract Syntax Tree. A tree representation of the structural logic of source code, commonly used in static analysis to examine how a program is built.
BLOB	Binary Large Object. A database data type used to store massive amounts of unstructured data, such as images, audio files, or videos.
CI	Continuous Integration. A software development practice where developers frequently and automatically merge their code changes into a central repository.
CLI	Command-Line Interface. A text-based interface used to interact with software by typing commands into a terminal.
Code Smell	A symptom in the source code of a computer program that possibly indicates a deeper problem or weakness in design, often used interchangeably with antipatterns.
Cohen's Kappa	A statistical measure used to calculate inter-rater (or intra-rater) agreement for categorical items, accounting for agreement occurring by chance.
Context Rot	A phenomenon where a Large Language Model's performance consistently degrades and becomes increasingly unreliable as the input context length grows.
CoT	Chain-of-Thought. A prompting strategy for Large Language Models that guides the AI to articulate its reasoning step-by-step before providing a final answer.
CSV	Comma-Separated Values. A simple text file format widely used to store and transfer tabular data.

CV	Coefficient of Variation. A statistical measure that represents the ratio of the standard deviation to the mean, used to show the extent of variability in relation to the average of the population.
DAO	Data Access Object. A software design pattern that isolates the database interaction logic from the rest of the application's business logic.
DBMS	Database Management System. A software application used to create, manage, and securely handle data within databases.
DDL	Data Definition Language. A subset of SQL used to create, modify, and define database objects like tables.
DML	Data Manipulation Language. A subset of SQL used for adding, updating, and deleting data within a database.
DQL	Data Query Language. A subset of SQL used for searching and retrieving specific data from a database.
DSL	Domain-Specific Language. A specialised programming language created for a specific problem or context (e.g., jOOQ's built-in query language).
Enum	Enumeration. A data type consisting of a set of named values, used in programming and database schemas to restrict a variable or column to predefined choices.
F1-Score	A machine learning evaluation metric that measures a model's performance by balancing both precision and recall into a single harmonic mean score.
Few-Shot Prompting	A technique where a Large Language Model is provided with a few examples of the desired task in the prompt to improve its output accuracy.
FN	False Negative. A detection error where an analysis tool fails to identify a flaw that actually exists in the code.
FP	False Positive. A detection error where an analysis tool mistakenly identifies a correct piece of code as flawed.
GUI	Graphical User Interface. A user interface that allows interaction with a program through visual elements like icons, buttons, and windows rather than text commands.
I/O	Input/Output. In the context of processing with language models, it refers to the data sent to the model (input tokens) and the response generated by it (output tokens).
IDE	Integrated Development Environment. A software application that provides developers with a comprehensive set of tools for writing, editing, debugging, and testing code.

IoU	Intersection over Union. An evaluation metric that measures the overlap between a predicted location and the actual target location, used in this thesis to assess antipattern localisation accuracy.
Jaccard Index	A statistical measure used to evaluate the similarity and diversity of sample sets. Applied in this thesis to measure the co-occurrence of different antipatterns.
JDBC	Java Database Connectivity. A standard Java API that enables Java applications to connect to and interact with relational databases.
jOOQ	jOOQ Object Oriented Querying. A Java library that allows developers to build and execute SQL queries in a type-safe, object-oriented manner.
JSON	JavaScript Object Notation. A widely used, lightweight data-interchange format that is easy for humans to read and for machines to parse.
JVM	Java Virtual Machine. System software that provides a runtime environment enabling computers to run compiled Java programs regardless of the underlying hardware.
LLM	Large Language Model. An artificial intelligence model trained on vast amounts of data, capable of understanding, analysing, and generating natural human language as well as programming code.
MENTOR	Measure, Explain, Nominate, Test, Optimize, Rebuild. A systematic framework and methodology used for evaluating and optimising the performance of database indexes.
ML	Machine Learning. A branch of artificial intelligence that enables systems to learn from past data and improve their performance on a specific task without being explicitly programmed.
Monte Carlo Method	A broad class of computational algorithms that rely on repeated random sampling to obtain numerical results, used in this thesis for balancing dataset splits.
NMS	Non-Maximum Suppression. An algorithm used to filter out overlapping predictions by selecting the most probable bounding box or code span and discarding the rest.
ORM	Object-Relational Mapping. A programming technique that links relational database tables to application objects, hiding direct SQL code generation from the developer.
PL/SQL	Procedural Language/Structured Query Language. Oracle Corporation's procedural extension for SQL and the relational database.
POJO	Plain Old Java Object. An ordinary Java object that is not bound by any special restrictions or framework-specific rules.

Prompt Engineering	The practice of designing and refining input instructions (prompts) to effectively guide Large Language Models toward generating specific, high-quality outputs.
Regex	Regular Expression. A sequence of characters that specifies a search pattern, often used for advanced text searching and manipulation.
SDK	Software Development Kit. A collection of software tools and libraries provided by hardware or software developers to build applications for specific platforms.
SotA	State-of-the-Art. The most recent, advanced, and highest-performing iteration of a technology or artificial intelligence model currently available.
Spearman's Correlation	A statistical measure used to evaluate the strength and direction of the relationship between the rankings of two variables, showing if they tend to increase together.
SQL	Structured Query Language. A standardised programming language used for managing relational databases and performing various operations on the data within them.
SQL Antipattern	A frequently occurring, seemingly effective but ultimately counter-productive approach to database design or querying that degrades system performance, maintainability, or data integrity in the long run.
TN	True Negative. A correct identification where the analysis tool accurately determines that a specific code segment contains no flaws.
ToT	Tree-of-Thought. A prompting strategy for Large Language Models where the model explores and evaluates multiple reasoning branches simultaneously, simulating a discussion among subject-matter experts.
TP	True Positive. A correct identification where the analysis tool successfully finds and marks an actually existing flaw in the code.
VAT	Value Added Tax. A consumption tax placed on a product or service, relevant in this thesis for accurate API cost calculations.
Zero-Shot Prompting	A technique where a Large Language Model is given a direct query or task instruction without any prior examples of correct answers, relying entirely on its pre-trained knowledge.

Table of contents

List of figures

List of tables

1 Introduction

SQL (Structured Query Language) is one of the most popular programming languages in the world [1], with the four most popular database management systems as of March 2026 all using SQL as their main language of interaction [1], [2]. At the same time, most developers using SQL have not been taught the best and worst practices of the language. They "are self-taught in SQL, learning it out of self-defence when they find themselves working on a project that requires it" [3, p. xvi].

This disparity—wide adoption with many users, but sparse systematic knowledge among them—results in a situation where the same mistakes are being repeated across the industry. When developers encounter problems with their database designs and queries, they often independently create solutions with similar flaws. These recurring missteps are known as SQL antipatterns—"the most frequent missteps software developers naively make while using SQL" [3, p. 1].

Previous research has shown that SQL antipatterns are prevalent among both industrial and open-source systems [4], [5, pp. 59–60], and they have quantifiable negative effects on the affected systems [6], [7, pp. 2334–2335]. Research has also shown that antipattern occurrences tend to remain unfixed for long periods of time and are likely never to be fixed at all [4]. The researchers suggested that this may be due either to developers' lack of awareness of SQL antipatterns and their disadvantages, or to the comparatively low priority assigned to fixing such antipatterns [4].

Both of these reasons could be attributed to the lack of SQL antipattern detection tools, which could bring attention to the occurrences of SQL antipatterns upon their introduction. As it stands right now, the tooling support for detecting SQL antipatterns in source code is scarce: while relevant tools exist for detecting antipatterns in plain SQL statements and some popular ORM (Object-Relational Mapping) frameworks, other frameworks and libraries lack support. One such library is jOOQ Object Oriented Querying (jOOQ)—a

popular Java library for building and executing SQL queries in a type-safe manner.

1.1 Research Objectives

The primary goal of this research is the development of a static analysis tool based on LLMs (Large Language Models), capable of detecting SQL antipatterns in jOOQ-based database access code.

We focus our research on LLM-based static analysis, rather than alternatives, because of their advanced understanding of code semantics and context, which is highly desirable in these circumstances. Some SQL antipatterns are highly context-specific, and detecting them in jOOQ-based code may require analysing multiple files and understanding the relationships between them. The jOOQ API (Application Programming Interface) is also extensive, and its dynamic nature allows SQL antipatterns to occur in many different forms, making comprehensive coverage through traditional static analysis very difficult.

Our secondary goal is to investigate the prevalence of SQL antipatterns in jOOQ-based code, as well as the ways in which these antipatterns manifest themselves. These results may help developers who use jOOQ become more aware of API usages that are strongly associated with antipatterns, thereby aiding them in *avoiding the pitfalls of database programming* [3]. Furthermore, these results can be used as input by the jOOQ maintainers when considering whether such APIs need clearer documentation or design changes in future versions.

We aim to answer the following research questions:

- **RQ1:** How do different LLMs compare in identifying SQL antipatterns in Java code that uses jOOQ for database access?
- **RQ2:** How do different prompting strategies affect the detection performance of LLMs in identifying SQL antipatterns in Java code that uses jOOQ for database access?
- **RQ3:** How accurately can the developed LLM-based tool detect SQL antipatterns in Java code that uses jOOQ for database access?
- **RQ4:** What patterns can be observed in the occurrence of SQL antipatterns in Java code that uses jOOQ for database access?

- **RQ5:** Which jOOQ API methods are most frequently associated with query antipattern occurrences?

1.2 Methodology and Contributions

This research follows the principles of Design Science Research [8]. The primary design artefact produced by this research is an LLM-based SQL antipattern detection tool for jOOQ.

To establish this tool's foundation, we compared multiple state-of-the-art LLMs and prompt engineering strategies (Zero-Shot, Few-Shot, Chain-of-Thought, Tree-of-Thought) using both quantitative metrics and qualitative inspection.

Additionally, we produced a human-labelled dataset of SQL antipattern occurrences in Java code, which uses jOOQ for database access, curated through repository mining and stratified random sampling. Moreover, we performed a large-scale analysis of the prevalence of SQL antipatterns in 602 existing Java software projects that use jOOQ.

To the best of our knowledge, this study is the first to evaluate LLM-based antipattern detection as a localisation task rather than a pure classification task. Instead of merely determining whether a file contains an antipattern, the proposed tool identifies the precise boundaries of antipattern occurrences.

Artificial intelligence-based tools were used as supportive aids in the research process and in the preparation of this thesis. The following tools were used to draft the following sections:

- **Google Gemini 3.1 Pro Preview [9]:** Abstracts, List of abbreviations and terms, Thesis Outline (Section 1.3), Reflection on Work Process (Section 8.2), Limitations (Section 8.3), Future Work (Section 8.4), Summary (Chapter 9).
- **Google NotebookLM [10]:** Logical Database Design Antipatterns (Section 2.1.1), Physical Database Design Antipatterns (Section 2.1.2), Query Antipatterns (Section 2.1.3), Application Development Antipatterns (Section 2.1.4).
- **OpenAI Codex [11]:** Development of Antipattern Detector (Chapter 5, introductory paragraphs), Workflow (Section 5.2).

OpenAI Codex was further used to develop the SQL antipattern detection tool (development workflow is elaborated on in Chapter 5), and to produce Figure 10. Gemini 3.1 Pro Preview and Gemini 3 Flash Preview [12] were further used extensively to improve wording in other sections.

We confirm that artificial intelligence was used strictly as a supportive aid. All generated content has been critically analysed, fact-checked, and edited by us to the extent necessary to ensure compliance with academic requirements. We take full responsibility for the accuracy, originality, and integrity of the work's content.

Unless otherwise specified, all data processing steps, experiments, and evaluations described in this thesis were implemented and executed within Jupyter [13] notebooks.

To ensure repeatability and facilitate future research, we open-sourced all artefacts produced as parts of the thesis. The finalised tool is available in the GitHub repository: https://github.com/kristoisberg/jooq-antipattern-detector. All other artefacts, including experimental scripts and their results, evaluated prompts, and source code for this document, are available in the following GitHub repository: https://github.com/kristoisberg/masters-thesis.

1.3 Thesis Outline

Chapter 2 provides the theoretical background on SQL antipatterns and the jOOQ library. We also review existing static analysis approaches and prompt engineering techniques used with large language models.

Chapter 3 details the methodology for creating a human-annotated ground truth dataset, encompassing the repository mining process, project sampling, and manual annotation guidelines.

In Chapter 4, we conduct an empirical evaluation of various prompting strategies, detailing how we measured the ability of several models to identify flawed database access code.

In Chapter 5, we provide an overview of the developed command-line application, covering the architectural design, internal workflow, and implementation technologies of the detection tool.

Chapter 6 outlines a framework for a large-scale project analysis, where we describe the statistical methods utilised to examine the prevalence and co-occurrence of antipatterns across hundreds of open-source repositories.

We present the answers to the formulated research questions in Chapter 7, highlighting the comparative performance of the evaluated models, the detection performance of our finalised tool, and the most common database missteps found in practice.

We then critically interpret these findings in Chapter 8, where we analyse the root causes of detection errors, compare our results with existing literature, and reflect on the methodology's limitations.

2 Background and Related Work

This chapter establishes the theoretical foundation for the thesis by introducing key concepts and reviewing related literature. We begin by detailing the history and impact of SQL antipatterns, followed by an overview of the jOOQ database access library. We then explore existing static analysis methods for code smell detection and discuss prompt engineering strategies used with large language models. Finally, we bridge these domains by reviewing previous efforts in automated SQL antipattern detection.

2.1 SQL Antipatterns

SQL antipatterns refer to frequently occurring counterproductive solutions in database schema design or query construction that initially appear effective, but ultimately result in poor performance, maintainability challenges, or compromised data integrity [3, p. 1], [14, p. 4]. The term "antipattern" was coined by Koenig in 1995 [15, p. 46] and is generally used to describe common solutions to recurring problems that prove to be ineffective and have negative consequences in the long term [3, p. 1], [16, p. 4], [17, p. 6]. A related term, "code smell", was coined by Beck and Fowler in 1999 [18, Ch. 3]. The term is used to describe the characteristics of source code that hint at the use of bad practices in the code [18, pp. 71–72].

SQL antipatterns were first described by Karwin in a book originally published in 2010 [3], which details "the most frequent missteps software developers naively make while using SQL" [3, p. 1]. Since the publication of the book by Karwin, researchers and industry professionals alike have produced numerous other collections and taxonomies of SQL antipatterns, both general [14], [19], [20], [21] and database vendor specific [22]. SQL antipatterns can have a negative impact on various characteristics of affected systems, such as performance, maintainability, portability, and data integrity [14, p. 4].

Although more extensive catalogues of SQL antipatterns have been produced, we limited

our efforts to the antipatterns defined by Karwin [3], which is considered a seminal work on the subject [4, p. 336] and have the most data available for comparison. Karwin is also working on a follow-up to his original book [23], containing 13 additional SQL antipatterns, expected to be published in June 2026 [24].

As an example, the "Implicit Columns" antipattern [3, Ch. 19] occurs when a `SELECT` statement does not specify the columns that are necessary to be selected, but rather selects them all using the $*$ wildcard. This degrades the maintainability of the affected system, since several database table operations (such as dropping a column) can change the ordinal positions of columns in the table. If code dependent on previously correct ordinal positions selects its data using the $*$ wildcard after the table layout has changed, it will receive columns in altered ordinal positions, resulting in faulty behaviour.

Furthermore, the antipattern hinders performance, since unneeded columns must be processed by the database management system (DBMS), and the corresponding data must be transferred to the affected system. This claim was confirmed by Lyu, Alotaibi, and Halfond [6, p. 60], who studied the performance impact of SQL antipatterns in the context of local SQLite databases of Android applications. They found that on average, fixing the antipattern reduced the runtime of affected queries by 27 %, and their energy consumption by 29 %.

The antipattern can also occur in the form of an `INSERT` statement, which does not specify the list of columns to which inserted values should be assigned, but rather depends on the implicit column order of the table [3, Ch. 19]. An example of an SQL query containing the "Implicit Columns" antipattern is shown in Figure 1.

```
SELECT * FROM employee WHERE name = 'Jack'
```

Figure 1. Query written in SQL containing the "Implicit Columns" antipattern.

Muse et al. [4] studied the prevalence of SQL antipatterns in data-intensive Java software systems and found that SQL antipatterns are prevalent in all studied application domains. For example, they found that the "Implicit Columns" antipattern affects nearly two queries out of every 100. Similarly, Sharma et al. [5, pp. 59–60] found that several database design antipatterns are prevalent in both industrial and open-source projects.

Muse et al. [4] also found that compared to traditional code smells [18, Ch. 3], SQL antipatterns tend to remain unfixed for a longer period and are more likely to persist throughout the lifetime of the software. They suggested that the reason for this could be developers' lack of awareness of SQL antipatterns and their downsides, or the comparatively low priority of the task of fixing antipatterns.

In the following sections, we provide a brief overview of each SQL antipattern defined by Karwin.

2.1.1 Logical Database Design Antipatterns

Logical database design involves developing solutions that remain agnostic of the implementation environment [25, p. 4]. Logical design antipatterns describe missteps in planning out database tables, columns, and relationships [3, p. 1].

- **Jaywalking [3, Ch. 2]:** Storing multiple values for a single attribute as a comma-separated string in one column to avoid creating an intersection table. This makes querying individual items inefficient and prevents the database from enforcing referential integrity.
- **Naive Trees [3, Ch. 3]:** Relying on a simple parent_id column to store hierarchical data, which makes it difficult to query deep or unlimited branches in a single step. Although easy to implement, it requires expensive recursive logic or many queries to retrieve full trees.
- **ID Required [3, Ch. 4]:** Using a generic, non-descriptive column named "id" as a primary key, or using a synthetic primary key, regardless of whether a natural or compound key would be more appropriate. This convention can lead to redundant or missing keys and allow duplicate rows in tables.
- **Keyless Entry [3, Ch. 5]:** Omitting foreign key constraints in an attempt to simplify design or improve performance, which forces the application code to manage referential integrity. This practice often leads to orphaned rows and corrupted data that must be cleaned up with periodic scripts.
- **Entity-Attribute-Value [3, Ch. 6]:** Using a generic table to store variable attributes as rows rather than columns to achieve an "open schema". This design breaks the relational model, making mandatory attributes impossible to enforce and significantly

complicating the creation of even simple reports.

- **Polymorphic Associations [3, Ch. 7]:** Using a multipurpose foreign key column to reference multiple parent tables, often distinguished by a "type" column. Because SQL foreign keys can only reference exactly one table, this design prevents the database from enforcing data integrity.
- **Multicolumn Attributes [3, Ch. 8]:** Creating multiple columns (e.g., tag1, tag2, and tag3) to store the values of a single multivalued attribute. This makes searching for values tedious and limits the number of items you can store to the fixed number of columns defined.
- **Metadata Tribbles [3, Ch. 9]:** Creating new tables or columns for each successive data value (such as a separate table or a column for each year) to manage large data sets. This leads to a proliferation of metadata objects that must be manually synchronised and updated as new data arrives.

2.1.2 Physical Database Design Antipatterns

Physical database design adapts logical designs for specific implementation environments, targeting them for particular hardware and software platforms [25, p. 4]. Physical design antipatterns describe missteps in defining tables and indexes, and choosing data types [3, pp. 1–2].

- **Rounding Errors [3, Ch. 10]:** Using the `FLOAT` or `REAL` data types for exact fractional values, such as currency, which results in cumulative precision errors during calculations. Using `NUMERIC` or `DECIMAL` types ensures values are stored and compared accurately.
- **31 Flavors [3, Ch. 11]:** Specifying a fixed list of allowed values directly in a column definition (using `CHECK` constraints or `ENUM`) instead of using a lookup table. This makes it difficult to update the allowed values or retrieve them for use in a user interface without changing the database schema.
- **Phantom Files [3, Ch. 12]:** Assuming that media files must be stored on the file system with only their paths saved in the database, which breaks transactional consistency. Storing media as `BLOB` data ensures that backups, deletions, and rollbacks are perfectly synchronised with the database.
- **Index Shotgun [3, Ch. 13]:** Creating database indexes by guessing rather than

following a plan, leading to either too many redundant indexes or none where they are actually needed. This can result in unnecessary overhead for updates or poor performance for critical queries.

2.1.3 Query Antipatterns

Query antipatterns describe missteps in manipulating and retrieving data with statements such as `SELECT`, `UPDATE`, and `DELETE` [3, p. 2].

- **Fear of the Unknown [3, Ch. 14]:** Treating NULL as an ordinary value or using ordinary values (like -1) to signify "unknown," leading to confusing results in logical comparisons. Since any comparison with `NULL` returns "unknown," queries using equality or inequality checking operators often fail to return expected rows.
- **Ambiguous Groups [3, Ch. 15]:** Referencing columns in a query with a `GROUP BY` clause that are neither grouped by nor part of an aggregate function. This can result in the database returning arbitrary and unreliable values for the non-grouped columns.
- **Random Selection [3, Ch. 16]:** Using `ORDER BY rand()` to fetch a random sample of data, which forces the database to perform an expensive table scan and manual sort. This technique is notoriously slow and fails to scale as the volume of data grows.
- **Poor Man's Search Engine [3, Ch. 17]:** Implementing full-text search using basic SQL pattern-matching predicates like `LIKE` or regular expressions instead of specialised tools. This approach cannot utilise standard indexes and often returns irrelevant matches that lack linguistic precision.
- **Spaghetti Query [3, Ch. 18]:** Trying to solve a complex, multistep problem in a single SQL statement to avoid running multiple queries. These convoluted queries are difficult to maintain and often produce unintended Cartesian products that multiply result sets incorrectly.
- **Implicit Columns [3, Ch. 19]:** Relying on wildcards like `SELECT *` or omitting column names in `INSERT` statements to reduce typing. This makes application code fragile, since it can break or reference the wrong data whenever the database schema changes.

2.1.4 Application Development Antipatterns

Application development antipatterns describe missteps in the use of SQL in the context of applications written in other programming languages [3, p. 2].

- **Readable Passwords [3, Ch. 20]:** Storing user passwords in plain text or with reversible encoding, exposing them to anyone with access to the database, logs, or backups. The secure solution is to store a one-way salted hash of the password that cannot be reversed by an attacker.
- **SQL Injection [3, Ch. 21]:** Building SQL queries by interpolating unverified user input directly into the query string, allowing attackers to manipulate the statement's syntax. Developers should use query parameters to ensure that user input is always treated as a literal value and never as executable code.
- **Pseudokey Neat-Freak [3, Ch. 22]:** Attempting to fill gaps in a primary key sequence or renumber rows to make them contiguous. This is unnecessary as surrogate primary keys are intended only as unique identifiers, and changing the values can break data integrity and relationships with other systems.
- **See No Evil [3, Ch. 23]:** Neglecting to check return values or handle exceptions from database API calls to keep the application code concise. Ignoring these signals leads to silent failures and makes it nearly impossible to diagnose simple syntax or connection errors.
- **Diplomatic Immunity [3, Ch. 24]:** Exemption of SQL and database code from standard software engineering practices such as version control, documentation, and automated testing. Treating SQL as a "second-class citizen" leads to high technical debt and a brittle, undocumented system.
- **Standard Operating Procedures [3, Ch. 25]:** Using stored procedures for all logic simply because "it's always been done that way", without evaluating modern architectural needs. This can concentrate the workload on a single database server and create bottlenecks, whereas modern application servers scale out more easily.

2.2 jOOQ Object Oriented Querying

jOOQ [26] is an open-source Java library that aims to simplify and enhance the way developers interact with SQL databases. Rather than abstracting SQL (like ORM frameworks

such as Hibernate [27] do), jOOQ integrates SQL into Java as an embedded domain-specific language (DSL) [28]. jOOQ allows developers to build and execute SQL queries using its DSL, while leveraging code generation to provide improved type safety and enhanced Integrated Development Environment (IDE) support [29]. Their code generator reverse-engineers an existing database schema into a set of Java classes that model numerous database objects, such as tables, sequences, stored procedures, and user-defined types [29]. When constructing SQL queries with jOOQ is not desirable, jOOQ also allows the execution of plain SQL statements through Java Database Connectivity (JDBC), serving as an object-oriented wrapper around the relatively low-level JDBC API [30].

A query written using the jOOQ DSL containing the "Implicit Columns" antipattern [3, Ch. 19], equivalent to the SQL query in Figure 1, is shown in Figure 2.

```
var employee = dslContext.select(DSL.asterisk())
    .from(EMPLOYEE)
    .where(EMPLOYEE.NAME.eq("Jack"))
    .fetchOne();
```

Figure 2. Query written in jOOQ containing the "Implicit Columns" antipattern.

The approach jOOQ takes to integrate SQL into Java makes it a popular choice for performing complex database interactions. It is one of the most widely used methods for interacting with databases in Java projects, having more than 6500 stars on GitHub as of October 2025 [31], and being widely adopted in industrial software [32]. It is widely used both as the sole method of database interaction and alongside ORMs such as Hibernate to implement the most complex and performance-critical database operations, which ORMs often struggle to execute efficiently, if they can execute them at all [33, p. 423].

2.3 Static Analysis for Antipattern Detection

Historically, the detection of antipatterns and code smells has been the domain of traditional static analysis methods, which take metrics-based and rule-based approaches, represented by tools such as SonarQube [34] and PMD [35].

In metrics-based approaches, code smells are identified based on threshold values defined for software metrics such as lines of code, complexity, and coupling [36]. For example, a class

with high complexity, low cohesion, and any direct access to foreign data could be identified as the "God Class", also known as "Blob" code smell [37, Ch. 3.3], [17, pp. 73–84], [36, p. 355].

In rule-based approaches, code smells are detected by searching the Abstract Syntax Tree (AST) of the source code for suspicious patterns [38]. For example, a class characterised by a procedural name, a very high number of lines of code, the absence of parameters, the use of global variables, and the lack of inheritance and polymorphism could be identified as exhibiting the "Spaghetti Code" smell [17, pp. 119–137]. This demonstrates a combination of rule- and metrics-based approaches [38, p. 27].

Although static analysis tools that use traditional methods have been effective in detecting code smells and antipatterns to an extent, they suffer from notable limitations due to the rigidity of the underlying methods. The metric- and pattern-based tools lack contextual awareness, making them prone to high false positive rates [39, pp. 10–11], often reporting issues, which are not actual problems upon human review, which reduces the practical usability of these tools and developers' confidence in them [40].

To overcome the limitations of rigid thresholds and patterns, researchers have explored machine learning (ML) techniques for code smell detection [41]. Rather than defining fixed thresholds or patterns, these approaches learn from data, such as human-annotated datasets of smelly and clean code [41]. Although recent approaches based on deep learning show promising results in terms of detection performance [42, pp. 3492–3494], traditional techniques remain prominent in real-world scenarios.

More recently, with the rapid evolution of LLMs, researchers are increasingly using them to detect problematic patterns in code. LLMs have repeatedly shown advanced semantic understanding and contextual awareness in software engineering tasks [43], [44]. Commercial tools such as CodeRabbit [45] are widely used to detect problems during code reviews, and various research has indicated that LLMs can detect code quality problems with reasonable precision [46], [47]. Most relevantly to our work, Sousa et al. [48, p. 410] found that both small and large language models are reasonably capable of detecting bad smells in PL/SQL code. It should be noted that the models used in many of these studies are considered outdated or deprecated by now, and repeating these studies with recent

models would likely demonstrate improved performance.

2.4 Prompt Engineering

The interaction between users and LLMs is mediated through prompting, which is the process of providing natural language inputs to guide the model towards generating a desired output [49]. Prompt engineering is the practice of intentionally designing, refining, and optimising prompts to guide LLMs to produce the most accurate, reliable, and useful outputs possible [49].

Effective prompt engineering involves the application of diverse prompting strategies—methodological frameworks that structure information, constraints, and instructions to better align the model's reasoning capabilities with specific task requirements [50].

The landscape of prompting strategies is vast and ever-changing, with a survey from 2024 [51] listing 41 distinct prompting strategies of varying levels of implementation complexity, used with different goals, such as eliciting reasoning and reducing hallucination. Some of the more popular prompting strategies include the following:

- **Zero-Shot Prompting:** Providing a model with a direct query or task instructions, without providing examples of correct answers in the prompt, leveraging the model's pre-existing knowledge and abilities to generalise and follow instructions [52]. Zero-Shot Prompting is the most common prompting technique and most representative of common LLM usage, and is often treated as the baseline prompting strategy.
- **Few-Shot Prompting:** Supplementing the prompt with a number of input-output examples, helping the model recognise patterns [53].
- **Chain-of-Thought (CoT) Prompting:** A collection of techniques used to guide the model to work through intermediate reasoning steps before arriving at a final answer [52], [54]. Although commonly considered a best practice, recent reports have indicated that its benefits have marginalised with the introduction of reasoning models [55]. We use the Zero-Shot version of CoT [52], appending the words "Let's think step by step." to our Zero-Shot prompt. Although there are more elaborate versions of CoT [54], their benefits have been shown to be similar to the Zero-Shot version [55, p. 3].

- **Tree-of-Thought (ToT) Prompting:** Expands on CoT by providing the model with an opportunity to explore multiple reasoning branches and to continuously self-evaluate by acting out a discussion between multiple subject-matter experts [56]. Applies concepts from more complex ToT frameworks [57], [58] at the prompt level.

2.5 Detection of SQL Antipatterns

To the best of our knowledge, there are currently (as of the beginning of the year 2026) two static analysis tools capable of detecting SQL antipatterns in Java code.

The first tool, *DbDeo* [59], was presented by Sharma et al. [5] and is capable of detecting nine schema-related antipatterns in SQL statements embedded in numerous programming languages. They employ regular expressions to extract SQL statements from code written in a host language. The extracted SQL statements are parsed and converted into models of the statements, which are then analysed by their smell detector. In their detection performance assessment, *DbDeo* demonstrated a low rate of false positives, all of which they attributed to errors during the extraction and parsing phases.

The second tool was presented by Nagy and Cleve [60] and is capable of detecting four of the query-related antipatterns described by Karwin [3]. Their tool uses intra-procedural string resolution to extract SQL statements from Java code, which are then parsed using a fault-tolerant SQL parser, capable of handling SQL statements with incomplete sections due to extraction errors. The tool can be accessed using a command-line interface or the *SQLInspect* plug-in for the Eclipse IDE [60], [61].

As the recommended method of constructing SQL statements with jOOQ is to use its DSL with code generation [62], most statements in projects using jOOQ are not in the form of plain SQL and the SQL is generated dynamically at runtime. Since both of the mentioned tools are designed to work with plain SQL statements, which must be present within the source code, they are incapable of detecting most SQL antipatterns in such projects.

Elsewhere, Almeida Filho et al. [63] used a tool named Manduka to detect bad smells in PL/SQL projects using static analysis, but the tool is no longer available. Following their footsteps, Sousa et al. [48] used language models to statically detect bad smells in PL/SQL projects.

Dintyala, Narechania, and Arulraj [7] presented *SQLCheck* [64], which is capable of detecting 26 different antipatterns using a combination of static and dynamic analysis. *SQLCheck* parses and analyses the query log of an application, searching for antipatterns both from individual queries and from the relationships of multiple consecutive queries. In their evaluation, *SQLCheck* achieved higher precision and recall than *DbDeo*.

As part of a series of articles [65], [66], [19], [20], Eessaar has published a collection of database queries for PostgreSQL, capable of detecting numerous database design problems by analysing the content of PostgreSQL system tables. Khumnin and Senivongse [67] have published a similar collection of queries for Microsoft SQL Server.

3 Dataset Creation

Because no public datasets of SQL antipatterns in jOOQ currently exist, this chapter details the methodology used to establish a ground truth. We describe the process of mining and filtering publicly available projects from GitHub, followed by the strategies used for sample generation, manual annotation, and splitting the data into training, validation, and test sets.

3.1 Repository Mining

We started by mining software repositories for projects, which we would later use for evaluation and analysis purposes. We wanted to collect as many projects as possible within reason, matching the following criteria:

- the project contains Java code that uses jOOQ for database access,
- the project stores jOOQ-generated classes as part of the source code, rather than treating them as derived artefacts, which are not put under version control [68].

The latter criterion was required to exclude projects that would otherwise have needed to be compiled in order to detect database design antipatterns. Due to the wide variety of Java versions, build tools, and configurations used in the projects, this would have required a large amount of manual effort on a project-by-project basis, which we opted against. For a considerable number of projects, this would have also been impossible for reasons such as private or otherwise unavailable dependencies, as well as jOOQ code generator configurations only capable of working on the original developer's machine.

To achieve this goal, we used the GitHub code search API [69], to search for projects that use jOOQ. The GitHub search API has a hard limit of 1000 results for each search term [69], which required us to minimise the number of duplicated results for each search term. For example, this meant that we could not search for import statements for jOOQ classes within Java files, since a few large projects could saturate the result list.

To find the largest number of unique projects possible, we limited our search to the manifest files of projects, looking for statements, which declare a dependency on jOOQ for the project. We limited the search to the manifest file formats of the two most popular build tools for Java [1]: Maven [70] (e.g., `pom.xml`) and Gradle [71] (e.g., `build.gradle` and `build.gradle.kts`). We designed the search terms to be as specific as possible to keep the number of results below 1000 when possible. For some search terms, the number of matching results exceeded the limit of 1000. In these cases, we retrieved the 1000 results that had been indexed most recently by GitHub's search infrastructure. The list of search terms used in the GitHub search API is provided in Appendix 2.

By performing the search on 10 January 2026, we identified **3829** projects that declared a dependency on jOOQ. We further filtered the gathered repositories by checking that the previously set criteria are met by making sure that the repositories contain jOOQ-generated classes as well as other Java files with references to jOOQ classes.

After excluding projects that did not store generated classes as part of their source code, we were left with **802** projects. Although not relevant to our research, an interesting observation can be made here: while both storing generated classes as part of the source code and treating them as derived artefacts are valid approaches with their merits and drawbacks [68], only 21 % of projects store them as part of the source code. We suspect that this is largely because the jOOQ code generator treats them as derived artefacts by default.

After excluding projects that did not contain any non-generated Java classes referencing jOOQ, we were left with **645** projects. The projects omitted by this step were mostly projects written in other programming languages running on the Java Virtual Machine (JVM), such as Kotlin and Scala.

Finally, we performed manual, LLM-assisted cleaning, with the aim of removing duplicate projects, which could later cause cross-contamination between training, validation, and test sets. We performed the analysis of duplicates with the Google Gemini 3 Pro Preview [72] model from its web interface. We provided the LLM with the file trees of the 645 remaining projects, prompting it to find potential clones and forks. We selected the model for its large context window, which was required to process the large collection of file trees,

combined with its high reasoning capabilities. Although the LLM did a remarkable job with the task, we manually reviewed the findings before any omissions.

For various reasons documented in Appendix 3, we omitted 42 projects. Most importantly, we found and removed 24 copies of a repository containing numerous Java- and Spring-related tutorials created by Baeldung [73]. While we also found a similarly sized cluster of projects created as part of a Java course by the Tinkoff bank in Russia, all of which had a similar structure and database schema, we decided to keep these projects, since each of them was created by a different author and had a distinct implementation and contained varying antipatterns. After the clean-up, we were left with **603** projects.

Later in the research process, preparing for the large-scale project analysis (Chapter 6), we discovered another large duplicated project, which would have skewed our analysis results. Hence, we later omitted that project from the analysis dataset, leaving us with **602** projects. The stages of mining and filtering projects are visualised in Figure 3.

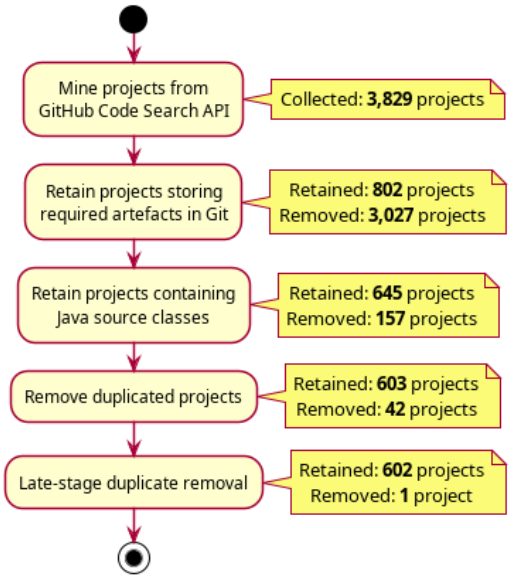

Figure 3. Project mining and filtering stages.

3.1.1 Considered Alternatives

Our method of mining repositories by using the GitHub code search API has the downside of being incomprehensive, meaning that it misses a portion of the projects available on GitHub for any of the following reasons:

- the project does not use a build tool,

- the project uses a build tool that is not Maven or Gradle,
- the project is **only** found by search terms, which return over 1000 results, and the project is omitted due to not being indexed recently.

Although the portion of missed projects is low (no other Java build tool is anywhere near the popularity of Maven and Gradle [1], and the more specific search terms should have caught all projects missed by the broader ones), it is a valid reason to mention alternative solutions we considered for mining Java repositories, and why we opted against using them.

Allamanis and Sutton [74, p. 209] presented a curated corpus of 14 807 open-source Java projects sourced from GitHub. As their work was published in 2013, it is outdated by 2026: a large part of the projects in the corpus has been deleted (as reported by Goeminne and Mens [75, p. 552] in 2015), new projects are missing, and the remaining projects could be using deprecated jOOQ APIs. Furthermore, their original corpus only contained ten projects using jOOQ [75, Fig. 1].

We considered creating an up-to-date version of the corpus by repeating the work of Allamanis and Sutton, processing GitHub's event stream, provided by GH Archive [76]. However, the dataset has grown rapidly since 2013, and combined with the suboptimal structure of the GH Archive dataset [77], it would have required us to commit an unreasonable amount of resources, given the preliminary nature of this research step.

Sharma et al. [5, p. 58] employed RepoReaper [78] to select open-source repositories for their study, finding 2568 high-quality projects in various programming languages, which contained SQL statements. However, RepoReaper used GHTorrent [79] to obtain metadata about GitHub repositories [78, p. 3223], and GHTorrent has not been updated since early 2021 [80]. Therefore, any set of projects produced based on GHTorrent would suffer from similar issues as the original corpus from Allamanis and Sutton.

Consequently, we found that our approach using the GitHub code search API is the best option available to us without committing unreasonable effort or resources.

3.2 Establishment of Ground Truth

To evaluate the detection performance of the prompting strategies and our later developed tool, we required a source of ground truth. As SQL antipatterns in jOOQ represent an unexplored area, to the best of our knowledge, no publicly available datasets of SQL antipatterns in jOOQ currently exist. Furthermore, the scientific literature provides no examples of how SQL antipatterns may occur in jOOQ, and such examples are also scarce elsewhere. For example, a remark in the jOOQ manual [81] shows that the "Implicit Columns" antipattern [3, Ch. 19] can occur in jOOQ in unexpected ways. Therefore, examples of antipatterns in plain SQL alone are insufficient for detecting SQL antipatterns in jOOQ.

To establish the ground truth, we manually annotated a subset of the projects gathered in Section 3.1.

3.2.1 Sample Generation

As the projects are of varying size, we wanted the subset to be representative of the complete set of projects regarding size distribution, avoiding bias towards small or large projects. Figure 4 displays the distribution of the number of relevant database access files per project across our dataset of 603 projects (still including the duplicated project that was filtered out at a later stage). We consider a Java source file to be a relevant database access file if both of the following conditions are met:

- it is a jOOQ-generated file or contains references to jOOQ or a generated file,
- it is not considered "irrelevant" by a manually compiled path list.

The path list consisted of jOOQ-generated files that were not useful for antipattern detection (e.g., records and POJOs), and files generated based on database entities not part of the project, such as system schemas of databases and tables managed by database extensions.

It should be noted that at this stage of the study, the list of "irrelevant" files was inconclusive and had some omissions, which we only discovered later. In Figure 5, we show the file distribution of 602 projects, omitting the later-discovered duplicated project, as well as using the final file list from a later stage of the study. It can be seen that, while the graph

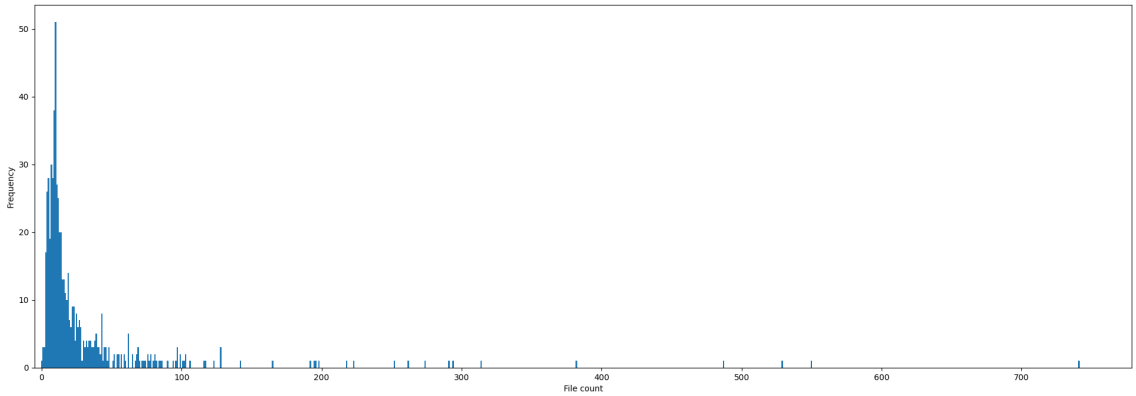

Figure 4. Distribution of relevant database access files per project.

shifts slightly, the overall distribution remains similar.

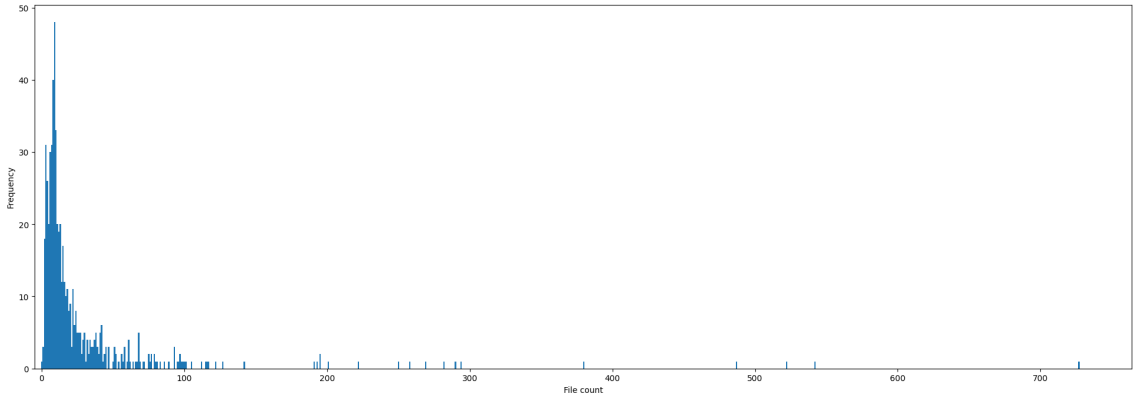

Figure 5. Distribution of relevant database access files per project (corrected).

As illustrated in the figures, the project distribution is positively skewed, with most projects clustered near zero (the "head") and a small number of substantially larger projects forming the "long tail". To ensure that small, medium, and large projects are adequately represented, we employed a stratified random sampling approach, selecting an equal percentage of projects from each group. Our initial plan was to select 5 % of the projects, but we later increased it to **10 %** due to an insufficient number of antipattern samples.

We divided the projects into size groups using the Head-Tail Breaks method. Given the set of projects as S and an individual project x, we calculated the two breakpoints, μ_1 and μ_2, which represent the upper file limits for small and medium projects:

$$\mu_1 = \frac{1}{|S|} \sum_{x \in S} x \tag{3.1}$$

$$S_{head} = \{x \in S \mid x > \mu_1\} \tag{3.2}$$

$$\mu_2 = \frac{1}{|S_{head}|} \sum_{x \in S_{head}} x \tag{3.3}$$

Based on these breakpoints, we classified the projects as:

$$\text{Class} = \begin{cases} \text{Small} & x \leq \mu_1 \\ \text{Medium} & \mu_1 < x \leq \mu_2 \\ \text{Large} & x > \mu_2 \end{cases} \tag{3.4}$$

We found the breakpoints to be **31** and **94**. Using these breakpoints, we classified 467 projects as small, 100 projects as medium, and 36 projects as large. Using the distribution from Figure 5, the numbers would have been slightly altered: the breakpoints would have been 29 and 91, and the class sizes would have been 468, 98, and 36.

Using the former classifications, we sampled 10 % of projects from each size class, resulting in 47 small, ten medium, and four large projects. In total, the sample of projects selected for annotation contained **61** projects. We used the fixed seed 123 456 for the pseudo-random number generator used for sampling, which ensures that the sampling process is consistent and reproducible.

3.2.2 Antipattern Selection

We limited the annotated antipatterns to the ones defined by Karwin [3]. Although we attempted to maximise the number of antipatterns included in our analysis and to interpret them as closely as possible to Karwin's definitions, we made several notable exclusions and interpretational deviations.

We excluded the "Index Shotgun" antipattern from our analysis. Karwin [3, pp. 147–152] recommends using the MENTOR (Measure, Explain, Nominate, Test, Optimize, and Rebuild) framework to choose optimal indexes. The framework involves measuring the runtime performance of SQL queries—a step that cannot be replaced by static analysis. Although other practitioners, such as Winand [82, pp. vi-vii], recommend creating optimal indexes during the development process, which could be aided by static analysis, choosing optimal indexes for a table can be a complex process with many trade-offs and deserves more dedicated research to be addressed appropriately.

We restricted our definition of the "Fear of the Unknown" antipattern to only include occurrences that are caused by flawed database design, as well as query conditions, which

do not contain values passed in from Java code. As the type system of Java is not null-safe, Java code can contain many potential nullability errors, but detecting them would have caused a large additional annotation burden. Furthermore, specialised tools (e.g., NullAway [83]) exist for detecting nullability errors in Java more efficiently and accurately.

Nagy and Cleve [60, p. 151] excluded the "Spaghetti Query" antipattern from their analysis, arguing that there is no objective definition to determine whether a query is a spaghetti query or not. They considered metrics, such as the length of a query in lines or characters, and counts of joins and subqueries, but found them too subjective.

Such metrics also do not tell the full picture, since one large query can often outperform multiple small queries due to optimisations in the database engine, and two queries with a similar length, involving the same number of tables, can differ vastly in their readability and performance. For example, Dombrovskaya, Novikov, and Bailliekova [84] describe several techniques for factoring large queries to improve their performance, readability, or reusability. Furthermore, the use of jOOQ DSL makes it possible to factor the source code of queries using object-oriented principles, further aiding with readability and reusability. For these reasons, we excluded the "Spaghetti Query" antipattern from our analysis.

We extended our definition of "Implicit Columns" to also include other types of blind projections, which select every column from a table, such as API methods, which fetch generated records from the database [81]. Such blind projections are considered a bad practice according to the jOOQ documentation, having similar negative effects to the cases described by Karwin [81].

We also excluded "Pseudokey Neat-Freak" and "Diplomatic Immunity" from our analysis, since they are rooted in organisational culture and cannot be detected from source code. Additionally, the use of jOOQ DSL largely mitigates the "Diplomatic Immunity" antipattern because SQL statements become first-class citizens of Java code.

Finally, we excluded the "See No Evil" antipattern due to difficulties in detecting it without performing runtime analysis. Unlike the Python examples provided by Karwin [3, p. 270], Java projects using modern application frameworks do not usually perform fine-grained database exception handling for each database-related statement, but apply a retry mechanism at a transaction boundary, or use global exception handlers (e.g., in

Spring [85] and Micronaut [86]), which could be implicitly installed by the framework. This makes accurate detection of the antipattern difficult for both human annotators and static analysis tools.

In total, we selected 19 antipatterns for annotation, including 8 logical design antipatterns, 3 physical design antipatterns, 5 query antipatterns, and 3 application development antipatterns. The list of included antipatterns by category is provided in Appendix 4.

3.2.3 Annotation Process

Based on prior experience with jOOQ and SQL antipatterns, we manually assigned labels to antipattern occurrences found in the 61 sampled projects by carefully manually reviewing their source code. We tracked the labels in a LibreOffice spreadsheet, which we later converted into a comma-separated values (CSV) file. For each antipattern occurrence, we marked down the following information:

- the type of antipattern,
- the project containing it,
- the path of the file containing it, relative to the root directory of the project,
- the start and end of the line span of the occurrence,
- a brief remark about the occurrence.

Due to the wide variety of Java versions, build tools, and local deployment methods used in the projects, along with occasional unavailable dependencies, we did not perform any runtime analysis on the projects.

As we studied a novel intersection between jOOQ and SQL antipatterns, not enough annotators sufficiently qualified in both domains were available to achieve a multi-annotator setup. Consequently, we employed a single-annotator setup, utilising the following means to maximise internal consistency.

Iterative Guideline Development

During the annotation process, we created a codebook for the annotation process in the form of flowcharts available in Appendix 5.

During the annotation process, we followed the flowcharts for each antipattern, annotating

their occurrences based on the visualised decision trees. When we spotted a variation of an antipattern or a non-antipattern edge case not yet covered by the corresponding decision tree, we adjusted the tree and re-reviewed previously, potentially incorrectly annotated files for consistency. This ensured that annotation decisions were made according to a consistent set of rules, rather than gut feeling.

Intra-Annotator Agreement

After a 30-day washout period, we re-annotated a subset of the previously reviewed files to verify the stability of the annotation logic over time. We sampled 10 % of the files containing at least one antipattern occurrence, and added an equivalent number of random files without antipattern occurrences to them. In total, we re-annotated 128 files using the same annotation guidelines. We used the fixed seed 123 456 for the pseudo-random number generator used for sampling, which ensures that the sampling process is consistent and reproducible.

To measure the agreement between the two sets of annotations, we employed Cohen's Kappa (κ). However, traditional Kappa is designed for nominal classification and does not account for the spatial displacement inherent in a multi-occurrence localisation task. Consequently, we adapted the metric to determine whether the two sets of annotations contain the same antipattern occurrences based on an exact spatial match.

We define a line span S as the set of line indices within an inclusive range in a source code file.

$$S = \{i \in \mathbb{N} \mid start \leq i \leq end\} \tag{3.5}$$

Given two annotators A and B, and their annotated line spans S_A and S_B, we consider the two line spans to match only if they cover the identical range of lines:

$$S_A = S_B \tag{3.6}$$

After aligning the spans using this exact-matching gate, we constructed a 16 × 16 confusion matrix whose rows and columns represented the 15 distinct antipatterns annotated across the 128 files, together with the absence of an antipattern. The complete matrix is provided in Appendix 6.

Given a set of k antipattern categories plus a "Background" category ($k+1$ total classes), let

n_{ij} denote the number of occurrences where Annotator A assigned class i and Annotator B assigned class j. The total number of unique candidate occurrences is $N = \sum_{i=1}^{k+1} \sum_{j=1}^{k+1} n_{ij}$.

The observed relative agreement (p_o) is defined as the proportion of occurrences where both annotators assigned the identical label to a spatially matching span:

$$p_o = \frac{1}{N} \sum_{i=1}^{k+1} n_{ii} \tag{3.7}$$

The probability of random agreement (p_e) is calculated based on the marginal totals of the confusion matrix. Let $n_{i\cdot}$ be the total number of times Annotator A used label i, and $n_{\cdot i}$ be the total number of times Annotator B used label i. The expected agreement is defined as:

$$p_e = \frac{1}{N^2} \sum_{i=1}^{k+1} n_{i\cdot} n_{\cdot i} \tag{3.8}$$

Where the marginal totals are derived as:

$$n_{i\cdot} = \sum_{j=1}^{k+1} n_{ij}, \quad n_{\cdot i} = \sum_{j=1}^{k+1} n_{ji} \tag{3.9}$$

Finally, Cohen's Kappa (κ) is computed to normalise the observed agreement against the possibility of chance:

$$\kappa = \frac{p_o - p_e}{1 - p_e} \tag{3.10}$$

We found the Cohen's Kappa to be **0.834**. According to the standard scale by Landis and Koch [87], this represents an almost perfect agreement between the sets of annotations, demonstrating that the resulting dataset is internally consistent. Despite this, we acknowledge that the lack of an inter-annotator agreement check ultimately leaves the dataset vulnerable to the systematic blind spots of a single annotator.

Results

In total, we annotated **1562** occurrences of 19 distinct SQL antipatterns within the 61 sampled projects. At least one occurrence of each antipattern was found in the sample, and every project in the sample contained at least one SQL antipattern occurrence. The most common antipatterns were "Implicit Columns", which we found 795 occurrences of in 59 projects, and "ID Required" with 259 occurrences in 49 projects. In Table 1, we show the prevalence of each SQL antipattern in the sampled projects.

Table 1. Prevalence of SQL antipatterns in sampled projects.

Antipattern	Occurrence Count	Containing Project Count	Occurrences per Project	Occurrences per Containing Project
Implicit Columns	795	59	13.03	13.47
ID Required	259	49	4.25	5.29
Keyless Entry	122	11	2.00	11.09
Fear of the Unknown	95	19	1.56	5.00
31 Flavors	71	11	1.16	6.45
Poor Man's Search Engine	55	12	0.90	4.58
Polymorphic Associations	40	1	0.66	40.00
Rounding Errors	39	7	0.64	5.57
Ambiguous Groups	24	2	0.39	12.00
Readable Passwords	14	9	0.23	1.56
Phantom Files	10	8	0.16	1.25
Naive Trees	10	4	0.16	2.50
Standard Operating Procedures	8	1	0.13	8.00
Jaywalking	6	2	0.10	3.00
Metadata Tribbles	5	3	0.08	1.67
Entity-Attribute-Value	3	3	0.05	1.00
SQL Injection	3	1	0.05	3.00
Random Selection	2	1	0.03	2.00
Multicolumn Attributes	1	1	0.02	1.00

3.2.4 Training-Validation-Test Split

We divided the set of annotated projects into the following three subsets:

- **Training set:** Used to continuously evaluate prompts during their development in Section 4.2. Also used as a source of examples for Few-Shot prompts.
- **Validation set:** Used to evaluate the detection performance of models and prompting strategies in Section 4.3.
- **Test set:** Used to evaluate the detection performance of the final tool in Section 5.6.

To avoid developing our prompts and evaluating them on files obtained from the same projects, we assigned projects to subsets as wholes, rather than splitting their files between different subsets.

We fixed the seed of the pseudo-random number generator used for splitting to ensure that the splitting process was consistent and reproducible, but we were unable to use the previously used fixed seed 123 456, since the produced splits were heavily unbalanced.

To achieve an optimally balanced data split, we employed a seed search based on the Monte Carlo optimisation method. We iterated over seeds in the range of 0 to 1 000 000, created a 34/33/33 training/validation/test split based on each seed, and calculated the sum of squared Coefficient of Variation (CV) for each split. We finally chose the split with the lowest sum of squared CV.

Given an individual antipattern x, a split i (assuming three splits), and the count of antipattern x occurrences in split i as $C_{i,x}$, we define CV for an individual antipattern as:

$$\mu_x = \frac{1}{3}\sum_{i=1}^{3} C_{i,x} \tag{3.11}$$

$$\sigma_x = \sqrt{\frac{1}{3}\sum_{i=1}^{3}(C_{i,x} - \mu_x)^2} \tag{3.12}$$

$$CV_x = \frac{\sigma_x}{\mu_x} \tag{3.13}$$

Then, given the set of antipatterns M, we performed the optimisation:

$$\Phi = \sum_{j=1}^{M} \left(\frac{\sigma(x_j)}{\mu(x_j)}\right)^2 \tag{3.14}$$

To ensure statistical validity, we applied filtering criteria before performing the search. We excluded all antipatterns that appeared in fewer than three projects or had fewer than 25 total occurrences. Consequently, 12 of the initially annotated 19 antipatterns were excluded due to insufficient data. All subsequent experiments, tool development, and large-scale analyses in this thesis focus exclusively on the remaining seven qualifying antipatterns (listed in Table 2).

During this step, we also reconsidered the sample size for Section 3.2.1, first increasing it from 5 % to 7 %, and then to 10 % of all projects, providing the search with a larger number of distinct project combinations and achieving a more balanced split.

The Monte Carlo optimisation-based search determined that the best possible split in this range was produced using the seed 767 573, which achieved a balance score of 0.52. The distribution of antipattern occurrences between the datasets can be seen in Table 2.

Table 2. Split of SQL antipattern occurrences between datasets.

Antipattern	**Total**	**Training**	**Validation**	**Test**
Implicit Columns	795	175	350	270
ID Required	259	55	99	105
Keyless Entry	122	30	56	36
Fear of the Unknown	95	23	36	36
31 Flavors	71	17	15	39
Poor Man's Search Engine	55	19	15	21
Rounding Errors	39	13	10	16

All produced datasets, along with their intended use cases later in the study, are visualised in Figure 6.

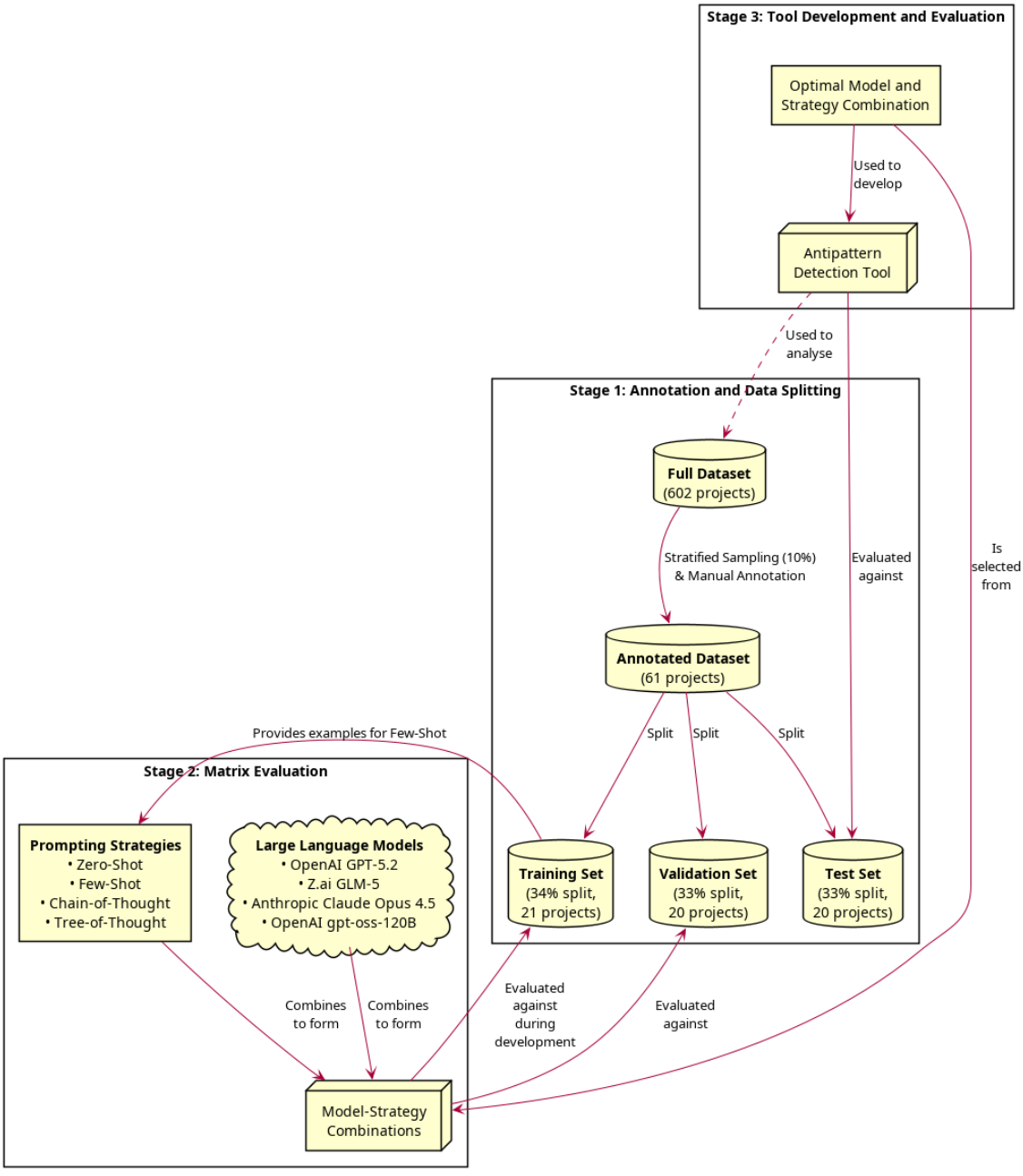

Figure 6. All created datasets with their intended uses.

4 Evaluation of Prompting Strategies

This chapter details the empirical evaluation of various prompting strategies for detecting SQL antipatterns in jOOQ. To assess the versatility and reliability of these strategies under different conditions, we benchmarked them across multiple contemporary LLMs, analysing both quantitative performance metrics and qualitative outputs.

4.1 Selecting Models

To assess the prompting strategies under varying conditions, we evaluated them against multiple models. We chose the language models based on criteria selected to aid us in the development and analysis process, while keeping the selection of models diverse. In particular, we searched for the following characteristics in the models:

- **Architectural diversity:** We aimed to include models with a diverse set of characteristics, such as different model sizes, reasoning vs non-reasoning models, and open vs closed-weight models.
- **Performance:** For a given architecture, the model should demonstrate near-state-of-the-art (SotA) performance on public benchmarks [88].
- **Repeatability:** The weights of the model should be stable, allowing the evaluation and analysis to be repeated in the future. Ideally, the model should have open weights. Alternatively, the model provider should provide point-in-time snapshots of the model.
- **Structured outputs:** The model should reliably support structured outputs, providing responses that reliably adhere to expected JSON Schema. This was desired to avoid the complexity of parsing non-structured outputs and handling of malformed responses. Structured output also increases determinism in responses by constraining decoding to outputs that satisfy the schema [89].

Based on this set of criteria, we settled on the following four models for our analysis:

- **OpenAI GPT-5.2 [90] (reasoning):** Near-SotA proprietary weights model. The best-performing model offering point-in-time snapshots.
- **Z.ai GLM-5 [91] (reasoning):** SotA open-weights model.
- **Anthropic Claude Opus 4.5 [92] (non-reasoning):** Near-SotA in models, which allow reasoning to be disabled. Superseded by Opus 4.6, which did not offer point-in-time snapshots at the time.
- **OpenAI gpt-oss-120B [93] (reasoning):** Open-weights, SotA in models classified as medium language models (up to 150B parameters).

Initially, we also planned to include Z.ai GLM-4.7-Flash [94], which was the best-performing small language model (up to 40B parameters) at the time, but we were required to omit it due to vast stability issues with all providers offering the model.

We most notably omitted Google Gemini 3.1 Pro Preview [9] and Anthropic Claude Opus 4.6 [95], which outperformed all the selected models, but due to their recent releases, did not have point-in-time snapshots available.

We were also not able to include any purely non-reasoning models in the comparison: such models have become increasingly rare, and the most capable purely non-reasoning model at the time, Kwaipilot KAT-Coder-Pro V1 [96], did not offer point-in-time snapshots. However, many modern reasoning models allow that capability to be disabled; therefore, we substituted the model with Anthropic Claude Opus 4.5, with the reasoning capability disabled.

We found that all selected models supported structured outputs with adequate reliability. In our preliminary experiments, some older models (e.g., Kimi K2 Thinking [97]) did not adhere to the provided JSON Schema, causing client-side validation to fail in numerous cases, but such issues were much less common in the selected models. The complete set of models considered for the analysis, along with their inclusion and omission reasons, is available in Appendix 7.

4.2 Designing Prompts

We designed prompts based on the four prompting strategies listed in Section 2.4. We limited our experiments to prompting techniques that could be fully executed in the span of

one prompt and one response to manage the scope of the thesis.

We iteratively developed and fine-tuned prompts for each technique, comparing the results against the training set produced in Section 3.2.4. We tested each iteration of our prompts against all tested models until all models reached near-optimal performance. To simplify the experimental setup and the interpretation of results, we only developed universal, model-agnostic prompts, without fine-tuning them for individual models.

Due to the high cost of LLM inference, we developed heuristics to target only the antipatterns that are potentially present in a given file. Firstly, we ignored files that did not contain any database-related code using the filters first mentioned in Section 3.2.1. Secondly, for each prompting strategy, we developed two prompts:

- for detecting query antipatterns from non-generated source files,
- for detecting (mostly) design antipatterns from files generated by jOOQ.

In addition to the file being analysed, we provided the design antipattern detection prompts with additional context by supplying them with a jOOQ-generated class containing the definitions of all primary, unique, and foreign key constraints present in that database schema. This additional context is used in the detection of the "Keyless Entry" antipattern to verify if there is an appropriate primary or unique key present, which the missing foreign key could reference.

We also applied the following pre-processing steps to the input files:

- To reduce the number of input tokens, and subsequently costs, we removed all leading and trailing whitespace from each line in the input files (including indentation).
- We noticed that the models consistently miscalculated the row spans of antipattern occurrences, if the analysed file contained two or more consecutive line breaks anywhere in the file before the antipattern occurrence. The models also often mislocated antipattern occurrences in large source files. To resolve these issues, we prefixed each line in the input file with the correct line number.

We designed our prompts to detect the seven qualifying antipatterns present in the training, validation, and tests sets created in Section 3.2.4. Designing them, we started with broad

instructions, allowing models to use their judgement and reasoning to a wide extent. As we identified instances where the model misinterpreted certain antipatterns or detected them incorrectly, we moved towards more concrete and specific instructions to eliminate ambiguity and ensure the model consistently adhered to our intended criteria.

Ultimately, it was necessary to define all detected antipatterns explicitly and, in certain cases, to specify the exact method calls and coding patterns to look out for. When models were provided only with the names of antipatterns, they tended to construct their own definitions, which were at times inconsistent with those proposed by Karwin [3]. We attribute this behaviour to the limited availability of literature on SQL antipatterns within the training data of LLMs, resulting in them "hallucinating" the definitions.

Furthermore, for the "ID Required", "31 Flavors", "Poor Man's Search Engine", and "Implicit Columns" antipatterns, it was necessary to provide explicit localisation rules to ensure consistent localisation behaviour.

For the Few-Shot Prompting technique, which required concrete examples, we acquired the examples from the training set produced in Section 3.2.4 by default. For antipattern variants and edge cases, which we were unable to find representative examples for from the training set, we created the examples synthetically.

We ultimately settled on the instructions detailed in Figures 7 and 8 for detecting antipatterns in generated and non-generated source files, respectively, consistent between all prompting strategies listed in Section 2.4. The complete prompts for all prompting strategies, containing sample input files, are shown in Appendix 8.

```
– ID Required: Never identify this issue in classes representing views,
  as views cannot contain primary keys. If the class represents a table,
  always detect the antipattern, if the name of the primary key is just
"id" (case-insensitive). Also detect the issue if a synthetic primary
key column exists, even though another unique constraint exists, which
is suitable as a primary key (the constraint is on columns, which are
virtually immutable by nature), and which does not complicate foreign
keys referencing the table too much. Only include the lines of the
primary key column definition in the line range, do not include
comments or anything else.
– Keyless Entry: A column, which refers to another table, is missing
its foreign key. Never identify this issue in classes representing
views, as views cannot contain foreign keys. Only report this issue if
```

```
the "Keys" class provided at the end contains a primary key that this
is appropriate for this column to refer to.
- Rounding Errors: Storing fixed-precision values in floating-point
type columns, such as FLOAT and REAL, rather than using fixed-precision
 types like DECIMAL and NUMERIC.
- 31 Flavors: Specifying allowed values in the column definition, i.e.
with a CHECK constraint or an ENUM type, rather than using a lookup
table. Only include the lines of the column definition in the line
range, do not include comments or anything else. Do not report the
issue if the CHECK constraint is used to check the value for emptiness
or against a range of values (including greater/lesser than comparisons
).
- Fear of the Unknown: A special default value, such as an empty string
, is used to mark a missing value, rather than NULL, and the special
value does not hold a semantic meaning. A column, which can never be
NULL in practice (e.g. it has a default value), is marked as NULLABLE.

If the file does not contain any antipatterns, leave the list of
occurrences empty.
```

Figure 7. Detection instructions used in DDL prompts.

```
- Poor Man's Search Engine: Usage of LIKE, ILIKE or regular
expressions to perform full-text search. Report the issue if it isn't
obvious from the method input parameters, whether the patterns contain
wildcards used for full-text search. Do not report the issue if LIKE,
ILIKE or regex is used for prefix search. Only include the line(s)
where the full-text search condition is created in the line range.
- Implicit Columns: A query fetching all columns from a database
table. In addition to obvious violations, report cases where jOOQ
fetches all columns of a table into records or generated DAOs (located
in a package ending with `tables.daos`). Do not report this issue if
it occurs within a `fetchCount` or `fetchExists` call. Only include
the line(s) where the blind projection is selected in the line range,
do not include the rest of the query.
- Fear of the Unknown: Query logic uses a NULLABLE column in a way
that produces incorrect results with NULL. Do not report issues that
arise from insufficient null-handling in Java code. Also do not report
the issue if you're unsure if the column is NULLABLE.

Only identify problems in code, which interacts directly with the jOOQ
DSL or generated DAOs (located in a package ending with
`tables.daos`). Do not identify problems in code, which interacts with
higher level abstractions. In case of multiple consecutive issues,
report them separately, even if they are on consecutive lines. If the
```

```
file does not contain any antipatterns, leave the list of occurrences
empty.
```

Figure 8. Detection instructions used in DML/DQL prompts.

4.3 Quantitative Analysis

Following the finalisation of the prompt design, we analysed each project in the validation set created in Section 3.2.4 using all combinations of models and prompting strategies. We subsequently evaluated the resulting outputs against the ground truth.

We performed the analysis using the OpenAI Python SDK [98], accessing the models through OpenRouter [99] as a unified gateway for different LLM providers, using the following parameters for LLMs:

- **Temperature**: Used to control the randomness of the output, with higher values resulting in less deterministic results. Using a value of zero encourages the model to select the most likely token, reducing randomness and improving repeatability, making the output as deterministic as possible. We used the default value 1.0 for OpenAI gpt-oss-120B, since using near-zero values caused it to repeat tokens infinitely, and this issue did not resolve with any other parameter changes. We used the value 0.0 for all other models, except for OpenAI GPT-5.2, which did not support the parameter when being used as a reasoning model.
- **Effort**: Used to control how much internal reasoning the model performs before generating a response. Higher effort levels enable deeper, multi-step analysis, generally improving reliability, while being slower, costlier, and less deterministic. We used the highest supported value for each reasoning model to leverage improved reliability, while compensating for decreased determinism with other techniques.

In Table 3 we show the selected parameter combinations for each model used for analysis.

Table 3. Model parameters used for analysis.

Model	**Temperature**	**Effort**
OpenAI GPT-5.2 (reasoning)	Unsupported	xhigh

Continues...

Table 3 – *Continues...*

Model	**Temperature**	**Effort**
Z.ai GLM-5 (reasoning)	0.0	Unsupported
Anthropic Claude Opus 4.5 (non-reasoning)	0.0	none
OpenAI gpt-oss-120B (reasoning)	1.0	high

While OpenRouter provides a convenient unified API, it acts as an abstraction layer where internal routing and caching semantics can vary. To maintain strict control over the experimental environment, we explicitly disabled OpenRouter's automatic model fallbacks. This ensured that requests were evaluated exclusively by the requested model or failed outright, triggering our local retry mechanism. We also considered OpenRouter's default request caching; however, because each analysed source file generated a strictly unique prompt and the evaluation was executed in a single pass, gateway-level caching could not artificially influence our results.

In essence, we were evaluating the performance of the models and prompts in performing a **multi-label** (there were multiple antipatterns that could be detected) **multi-occurrence** (each antipattern could occur multiple times within each file) **localisation** (in addition to classification, we aimed to detect the location of the antipattern occurrences) task. For this reason, we employed a combination of spatial overlap criteria and classification metrics for evaluation, using Intersection over Union (IoU) and F1-score.

The first step involved quantifying the localisation performance using IoU. We define a line span S as the set of line indices within an inclusive range in a source code file.

$$S = \{i \in \mathbb{N} \mid start \leq i \leq end\} \tag{4.1}$$

For a predicted line span S_p and a ground truth line span S_g, the IoU is defined as:

$$\text{IoU} = \frac{|S_p \cap S_g|}{|S_p \cup S_g|} \tag{4.2}$$

This metric produces a value between 0 and 1, where 1 represents an exact line-level match. In this methodology, the IoU serves as the qualification gate for all subsequent metrics. A prediction is classified as a true positive (TP) if its spatial overlap with the ground truth

exceeds the threshold: IoU ≥ 0.5. If the IoU is below this threshold, the prediction is classified as a false positive (FP), while missed ground truth spans are considered false negatives (FN).

Using this qualification gate, we classify the predictions into TPs, FPs, and FNs, employing Non-Maximum Suppression (NMS): in cases of multi-occurrence where multiple predictions overlap a single ground truth span, only the prediction with the highest spatial overlap is assigned as a TP, while subsequent overlaps are penalised as FPs.

Based on the classifications, we calculated the precision (P) and recall (R) of the model at the selected threshold, defined as:

$$P = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad R = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4.3}$$

Precision measures the exactness of the model by calculating the ratio of correctly identified detections to the total number of detections, thereby indicating the extent to which the model avoids FPs. Recall measures the completeness of the model by calculating the ratio of correctly identified detections to the total number of antipattern occurrences in the ground truth, reflecting the model's ability to minimize FNs.

Using these values, we calculated the F1-score, which provides a balanced assessment of model performance at the selected threshold by calculating the harmonic mean of precision and recall, serving as a single unified metric that penalizes extreme values in either component:

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{4.4}$$

We calculated P, R and F_1 for each antipattern separately, and as a weighted average of all antipatterns:

$$\bar{x}_w = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i} \tag{4.5}$$

This way, we were provided with an understanding of which models and prompts are the best in the detection of antipatterns in general, but we also had the opportunity to study the differences in further detail. Additionally, we recorded the cost and runtime of each analysis and the number of requests retried due to incorrectly formatted (e.g., empty or non-JSON) responses provided models.

We acquired the data about costs from the OpenRouter dashboard due to a subtlety in the OpenAI Python SDK's API for structured outputs: if the client-side validation for the output fails, the request, including its usage data, cannot be obtained from the client code, making accurate cost calculations impossible. We represent the cost in USD including VAT, and the runtime in seconds.

Based on the results of this analysis, we could provide answers to **RQ1** and **RQ2** in Sections 7.1 and 7.2, and we could decide on the preferred prompting strategy for the development of an antipattern detection tool in Chapter 5.

4.4 Qualitative Analysis

We manually inspected the results produced by each combination of model and prompting strategy. Where possible, we also inspected the reasoning data to identify behavioural patterns that could explain the frequent FPs and FNs in the results. We then compared our experimental findings with existing literature on prompting strategies and LLM behaviour.

We discuss the results of the qualitative analysis in Section 8.1.1.

5 Development of Antipattern Detector

This chapter provides an overview of the developed command-line tool for detecting the seven targeted SQL antipatterns, which employs the most effective prompting strategy identified based on the analysis that was described in Section 4.3. We discuss the tool's architectural design, internal workflow, and configuration options, as well as the coding process used to implement it.

The tool was developed with the assistance of OpenAI Codex [11], using the GPT-5.4 [100] model, in a semi-autonomous agentic coding process. The development process was organised around iterative implementation tasks. For each task, we specified the intended behaviour, constraints, and expected outcome, after which the coding agent inspected the existing codebase, proposed or implemented the necessary changes, and ran relevant checks where applicable. Human supervision was retained throughout the process: generated changes were reviewed, requirements were clarified when necessary, and architectural decisions, evaluation requirements, and final acceptance of the implementation remained under human control. In this way, Codex was used primarily as an implementation assistant, while the design goals and validation criteria were defined by us.

5.1 Architecture

We implemented the tool as a monolithic command-line interface (CLI) application. Compared to a graphical user interface (GUI) application, this approach offers greater versatility, allowing the tool to be run in non-interactive environments, such as continuous integration (CI) pipelines and batch evaluation scripts. Additionally, the CLI approach avoids coupling the core analysis logic to a particular user interface. This allows for developing alternative frontends in the future, such as a standalone desktop application or an IDE extension, without requiring substantial changes to the analysis pipeline.

The tool is built as a self-contained executable without local runtime dependencies (e.g.,

Python or Node.js), simplifying distribution and use in controlled evaluation environments. It features extensive configuration options (detailed in Section 5.5), which can be provided as command-line options, environment variables, or in a configuration file. In case of duplicated options, command-line options take precedence over environment variables, which in turn take precedence over the configuration file.

A high-level diagram of the tool's architecture is shown in Figure 9.

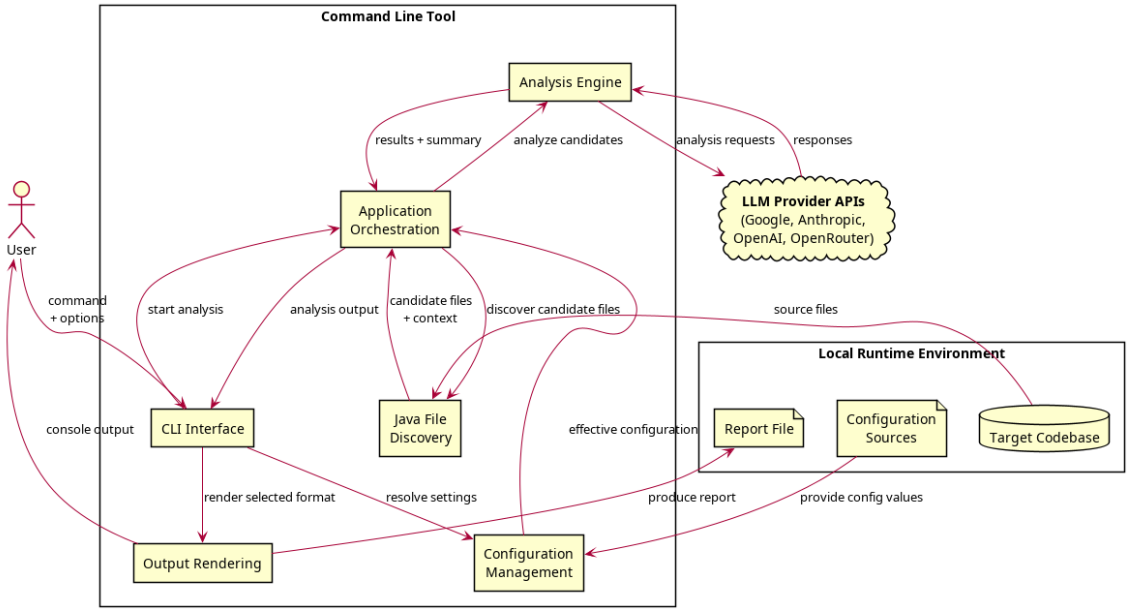

Figure 9. Components of the SQL antipattern detection tool.

5.2 Workflow

Although the tool is implemented as a monolithic application, its internal architecture is organised as the staged pipeline shown in Figure 10. The main stages of the pipeline are the following:

- **Input and configuration:** The tool receives command-line arguments, validates the input directory, and combines command-line options, environment variables, an optional configuration file, and built-in defaults into a single validated runtime configuration.
- **Model setup:** The selected model is resolved from an explicit provider-prefixed identifier. The provider prefix determines which LLM client is created and which API key is required.
- **Source-file discovery:** The selected project directory is recursively scanned for Java

source files. The tool then applies jOOQ-oriented inclusion and exclusion heuristics to retain only files likely to contain relevant schema or query logic.

- **Candidate preparation:** Each retained Java file is converted into an analysis candidate containing its path, contents, analysis category, and optional contextual information. The analysis category determines whether the file is analysed as schema-definition code or query/application code. The tool also attaches the closest available key-definition context when such context is found.
- **Per-file analysis:** Candidate files are processed concurrently up to the configured concurrency limit. For each candidate, the source code is converted into a line-numbered representation and inserted into an LLM request. When the prompt-size budget requires it, the source context and any attached key-definition context are truncated while preserving useful context where possible.
- **Validation and failure handling:** If a request fails or returns data in an unexpected format, the tool retries it according to the configured retry count. If all attempts fail, the error is reported as a per-file result instead of aborting the whole run.
- **Aggregation and rendering:** After all candidates have been processed, the tool sorts the per-file results, computes a run-level summary, and renders the structured result as text, JSON, or CSV. The rendered output is then written either to standard output or to a configured output file.

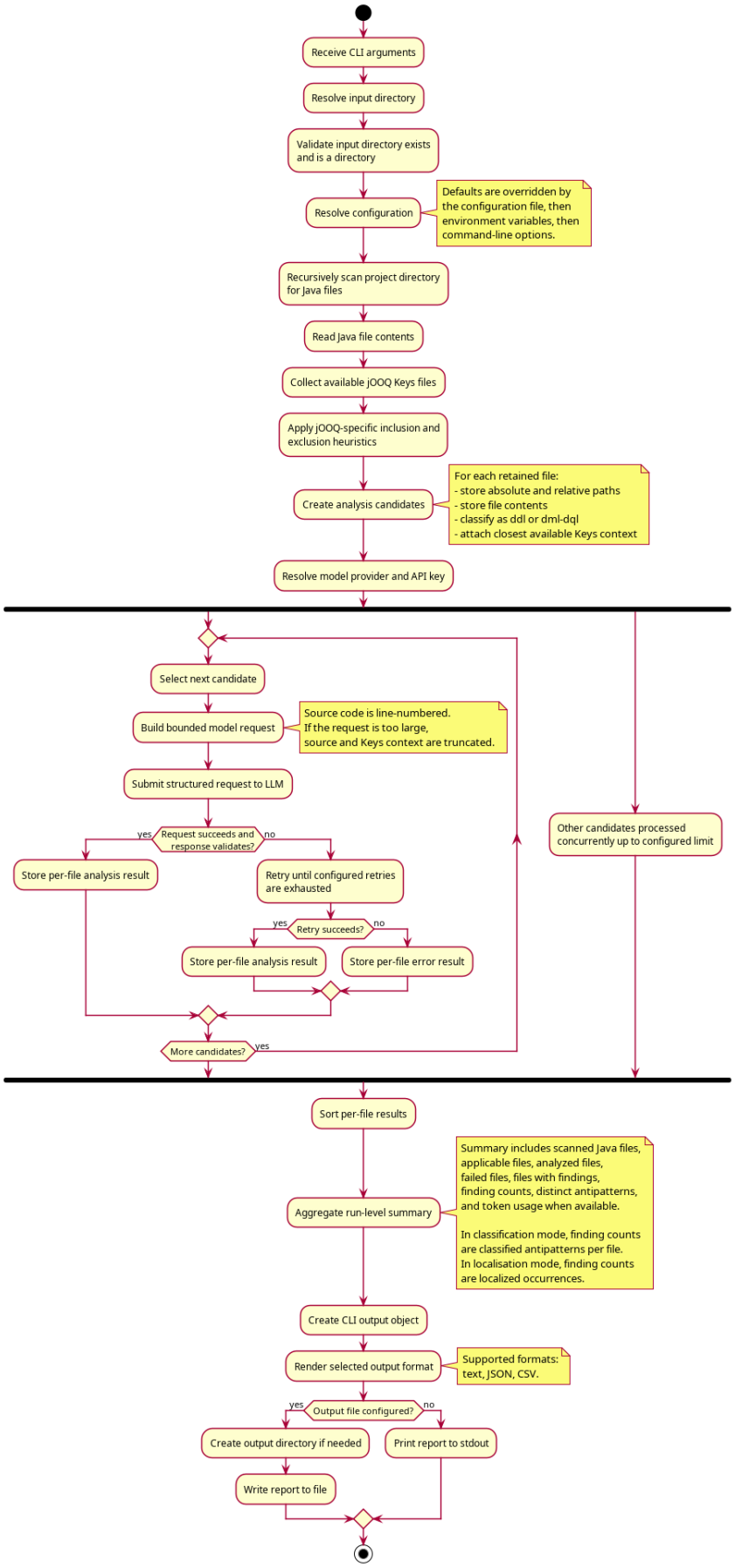

Figure 10. Workflow of the SQL antipattern detection tool.

5.3 Result Representation

The tool supports two analysis modes. In localisation mode, the output contains occurrence-level findings with source locations and explanations. In classification mode, the output contains only the distinct antipatterns detected in each file. The output of a run is a structured report consisting of run-level metadata, aggregate statistics, and per-file analysis results. The text rendering is intended for human inspection, while JSON and CSV outputs support automated processing, evaluation scripts, and integration with other tools.

Figure 11 shows a shortened text rendering of an analysis report in the localisation mode. The report begins with the selected model and analysed directory, followed by aggregate statistics. Individual findings are then grouped by file and include the detected antipattern, source location, relevant code fragment, and an explanation.

```
jOOQ Antipattern Detector
Model       google:gemini-3.1-flash-lite-preview
Directory   /home/kristo/masters-thesis/sample-project

Summary
Scanned Java files        164
Applicable files           39
Analyzed files             39
Failed files                0
Files with findings         7
Total occurrences          14
Distinct antipatterns       2
Input tokens            68739
Output tokens            1925
Total tokens            70664

Findings
  ------------------------------------------------------------

src/main/java/com/inventory/prosta/bot/jooq/tables/AccountChat.java
  ID Required (54-54)
  Code        public final TableField<AccountChatRecord, Long> ID =
  createField(DSL.name("id"), SQLDataType.BIGINT.nullable(false).
  identity(true), this, "");
  Explanation The primary key is named 'id', which is a generic name
  that provides no semantic information about the entity. Consider
  renaming it to 'account_chat_id' to improve clarity and consistency
  across the database schema.

  ------------------------------------------------------------

src/main/java/com/inventory/prosta/bot/repository/AccountRepo.java
```

```
Implicit Columns (64-64)
Code        dsl.select()
Explanation Using select() without arguments fetches all columns from
 the table, which can lead to performance issues and unnecessary data
 transfer. Explicitly list only the required columns in the select
clause.

Implicit Columns (90-90)
Code        dsl.select(ACCOUNT.fields())
Explanation Using ACCOUNT.fields() fetches all columns from the table
, which is inefficient if only a subset of data is needed. Specify
only the necessary columns to improve query performance and
maintainability.
```

Figure 11. Results displayed by the tool (truncated).

5.4 Implementation Technologies

The tool utilises the following technologies:

- **Bun [101]:** Modern JavaScript toolkit, consisting of a JavaScript runtime, bundler, test runner, and package manager. It was selected for its high performance, simple setup, built-in support for TypeScript, and built-in ability to compile applications into standalone executables.
- **TypeScript [102]:** Strongly typed superset of JavaScript, enabling developers to catch errors (e.g., type mismatches and nullability errors) during development, rather than at runtime. It was selected for its compile-time type safety and enhanced IDE support compared to JavaScript.
- **Biome [103]:** Modern code formatter and linter for JavaScript and TypeScript. It was selected for its high performance, simple setup, and built-in support for TypeScript.
- **Vercel AI SDK [104]:** Standardised interface for communicating with LLM providers, hiding provider-specific interface differences behind a unified abstraction.
- **Commander [105]:** Lightweight framework for building CLI applications, used for parsing command-line arguments.
- **fast-glob [106]:** Performance-oriented library for traversing the file system and retrieving file paths based on "glob" patterns (e.g., `**/*.java`). Used to find relevant files in the directory of the analysed project.

- **p-limit [107]:** Library for running multiple asynchronous functions with limited concurrency. Used to perform a limited number of parallel requests against the LLM provider.
- **Zod [108]:** TypeScript-first library for schema validation. Used for defining schemas for configuration options and LLM responses, and for validating objects against their respective schemas.

5.5 Configuration Interface

The configuration interface was designed to support both interactive use and automated evaluation runs. The available options can be grouped into five categories: model selection, analysis behaviour, execution control, output control, and provider authentication. This made it possible to reuse the same implementation for manual inspection, quantitative experiments, and automated result collection.

Although the tool provides user-friendly defaults for common command-line usage, most of the execution parameters can be overridden to support controlled experiments. For example, the output format can be changed from human-readable text to JSON or CSV, allowing results to be consumed by evaluation scripts or third-party tooling. Similarly, while the default prompts are embedded within the executable binary to keep the tool fully self-contained by default, they can be loaded from a JSON file with an execution parameter, enabling the use of other prompting strategies or the detection of additional antipatterns.

A comprehensive overview of all available configuration options, including their accepted parameters and default values, is provided in Appendix 9.

5.6 Quantitative Analysis

To validate the detection performance of the finalised tool, we performed an analysis on each project in the test set produced in Section 3.2.4, using the tool in the default "localisation" mode. As in Section 4.3, we evaluated the tool using metrics suitable for a multi-label multi-occurrence localisation task: IoU and F1-score.

So far, we have treated the task as a localisation task, and validating it as such; however, there is not much comparison data available for such an approach. To compare our results

against other works regarding detecting antipatterns and code smells using LLMs, we also evaluated it as a multi-label classification task.

To achieve this, we re-analysed the projects in the test set, this time using the tool in the "classification" mode, and compared the results against the ground truth. In this case, a prediction was considered a TP, if the tool detected an antipattern within a file, and the ground truth also contained the same antipattern within that file.

Based on the results of the quantitative analysis, we provide an answer to **RQ3** in Section 7.3. We compare our performance metrics against previous works regarding both SQL antipattern detection and LLM-based code smell detection in Section 8.1.3.

5.7 Qualitative Analysis

We manually inspected the differences between the tool's results and the ground truth, seeking patterns within the FPs and FNs and discussing their root causes in Section 8.1.2.

We also compared the outputs of the "localisation" and "classification" modes. The differences in the results were analysed to identify common localisation errors and their underlying causes, and to explain why classification metrics may yield stronger results than localisation metrics.

6 Project Analysis

This chapter outlines the framework for our large-scale empirical analysis, which applies the newly developed tool to the complete dataset of **602** mined software projects. We detail the statistical methods and metrics used to examine the prevalence and co-occurrence patterns of the seven evaluated SQL antipatterns in real-world projects.

6.1 Quantitative Analysis

For each of the seven targeted antipatterns, we calculated the following statistical values:

- the total number of detected antipattern occurrences,
- the number of projects containing that antipattern,
- the average number of detected occurrences per project among all 602 projects,
- the average number of detected occurrences per project among projects containing that antipattern,
- the number of detected occurrences per 100 jOOQ statements among all 602 projects.

For antipatterns, which have been studied by other works, we compared our results to theirs. Other works [4], [5], which detect SQL antipatterns by extracting SQL statements from source code and parsing them, receive the number of SQL statements in the project as a side product of the extraction step. However, since the jOOQ API allows SQL statements to be constructed dynamically, it is difficult to accurately estimate the number of SQL statements in the projects.

Therefore, rather than estimating the number of SQL statements in the projects, we estimate the number of jOOQ statements as the number of references to jOOQ API methods that execute an SQL statement. It should be noted that because loop constructs and conditional query building can generate a vastly different number of actual SQL statements at runtime than static method references imply, this proxy measurement introduces a margin of error. As a result, the numbers between our study and others are not directly comparable. Despite

this, they can be used as indications of broader trends.

To study the co-occurrence patterns of SQL antipatterns, we calculated the Jaccard Indexes, Conditional Probabilities, and Spearman's correlations between the antipatterns, both at the project and file level.

The Jaccard Index measures the degree of co-occurrence between two SQL antipatterns, allowing us to identify "antipattern clusters"—groups of antipatterns that consistently occur together. A value of 1.0 indicates that two antipatterns always appear together, while 0.0 indicates that they never appear in the same project or file.

Given A and B, which represent the sets of projects or files containing each antipattern, and $|A|$ and $|B|$, which represent the number of observations in those sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{6.1}$$

Conditional Probability measures the directional dependency between two SQL antipatterns. It calculates the likelihood that antipattern B is present, given the presence of antipattern A. This metric allows us to identify "indicator" antipatterns, which may predict the presence of others.

$$P(B|A) = \frac{|A \cap B|}{|A|} \tag{6.2}$$

Spearman's correlation (ρ) measures the relationship between the intensity of two antipatterns, rather than just their presence. Whereas the other metrics merely indicate the presence of a smell, Spearman's correlation incorporates occurrence frequencies to assess whether increases in one antipattern are associated with increases in another.

To account for tied ranks, which occur frequently when multiple projects share the same antipattern count, a correction factor T is applied to the variance. This factor is calculated for each antipattern based on the number of tied groups G and the number of tied observations t_j within the j-th group:

$$T = \sum_{j=1}^{G} \frac{t_j^3 - t_j}{12} \tag{6.3}$$

Given d_i, which represents the difference between the ranks of frequencies of the two antipatterns in a project or file i, and n, which represents the total number of projects or files containing at least one of the antipatterns:

$$n = |A \cup B| \tag{6.4}$$

$$\rho = \frac{\frac{n^3-n}{6} - \sum d_i^2 - T_A - T_B}{\sqrt{\left(\frac{n^3-n}{6} - 2T_A\right)\left(\frac{n^3-n}{6} - 2T_B\right)}} \tag{6.5}$$

Based on these results, we provide an answer to **RQ4** in Section 7.4.

For two of the detected query antipatterns ("Implicit Columns" and "Poor Man's Search Engine"), we analysed the jOOQ API methods associated with each antipattern occurrence. We employed an iterative pattern-matching approach, assigning a label to each antipattern occurrence by comparing the containing code snippet against a collection of patterns, each of which indicated an associated API pattern for the antipattern occurrence.

We skipped the analysis for all database design antipatterns, since for them, the associations only display how jOOQ generates classes based on an underlying problematic schema, and do not provide insights into potential API design issues.

We also skipped the analysis for the "Fear of the Unknown" antipattern, which we detect from both classes representing the database schema and source classes containing DQL and DML statements. We initially planned to constrain the labelling to only occurrences found in DQL and DML statements, but only a negligible number of "Fear of the Unknown" occurrences (16 occurrences, 1.3 % of total) were found in DQL and DML statements, not providing a sufficient sample size.

Based on this analysis, we provide an answer to **RQ5** in Section 7.5. We developed the pattern collection iteratively until we were able to assign a label to at least 98 % of all antipattern occurrences, at which point only uncommon occurrences were left unlabelled, which did not affect the answer to the research question.

6.2 Qualitative Analysis

We discussed the results of quantitative analysis from a qualitative perspective in Sections 8.1.3 to 8.1.5. We assessed whether there were any major discrepancies between the prevalence numbers found by us and prior works focusing on plain SQL, and discussed the reasons behind them.

Additionally, we examined the most prominent co-occurrence patterns between antipatterns, discussing their implications, as well as the jOOQ API methods most commonly associated with SQL antipatterns.

7 Results

This chapter presents the empirical results of our research, directly answering the five research questions formulated in Section 1.1. We report on the comparative performance of the evaluated LLMs and prompting strategies, the detection performance of the finalised detection tool, and the estimated prevalence of SQL antipatterns. Further interpretation of these findings is reserved for Section 8.1.

7.1 RQ1: How do different LLMs compare in identifying SQL antipatterns in Java code that uses jOOQ for database access?

To evaluate the baseline performance of LLMs in detecting the seven qualifying SQL antipatterns, we used the performance of the Zero-Shot Prompting strategy, which uses the fewest prompt engineering methods and is the most representative of real-life use. In Table 4 we show the baseline detection performance of SQL antipatterns for each compared LLM, calculated as the weighted average of the detection performances of individual antipatterns. In Appendix 10 we show the baseline detection performances for individual SQL antipatterns.

Table 4. Binary evaluation metrics, costs, runtimes, and retry counts for detection of SQL antipatterns using Zero-Shot Prompting.

Model	**Precision**	**Recall**	**F1-Score**	**Cost (USD)**	**Runtime (s)**	**Retries**
OpenAI GPT-5.2 (reasoning)	0.85	0.92	0.88	26.16	2423	1
Z.ai GLM-5 (reasoning)	0.88	0.89	0.88	11.77	5774	175
Anthropic Claude Opus 4.5 (non-reasoning)	0.88	0.89	0.88	29.97	367	0
OpenAI gpt-oss-120B (reasoning)	0.83	0.87	0.83	1.49	2808	0

All near-SotA models (GPT-5.2, GLM-5, and Opus 4.5) achieved a weighted average F1-score of 0.88, demonstrating strong detection performance. The difference lies in the

paranoia of the models: GPT-5.2 achieved a greater recall than the other models (0.92 vs 0.89), but a lower precision (0.85 vs 0.88), capturing more TPs, while also returning more FPs.

However, the models achieved strong performance at significantly different costs and runtimes. As is common for non-reasoning models, Opus 4.5 was the fastest but also the most expensive, completing the analysis of 20 projects (containing 823 analysed "relevant" files) in six minutes and seven seconds at a cost of $29.97. GLM-5 was the opposite: nearly three times cheaper at $11.77, but also over 15 times slower due to stability issues explained later. GPT-5.2 was in the middle, costing slightly less than Opus 4.5 at $26.16, but being 6.6 times slower due to its heavy reasoning process. The gpt-oss-120B model was by far the cheapest, costing just $1.49 while being slightly slower than GPT-5.2.

An interesting difference between the near-SotA models was that Opus 4.5 was consistently superior in the precision of detecting the "Keyless Entry" antipattern, achieving a precision of at least 0.72 in all prompting strategies, also consistently displaying a very high recall of at least 0.98, while all other models never exceeded a precision of 0.64.

It should also be noted that for both the baseline prompting strategy and all tested prompting strategies, GLM-5 and gpt-oss-120B models showed unstable behaviour. The GLM-5 model was by far the slowest with inconsistent runtimes and had a tendency to return misformatted and empty responses, causing the analysis for these files to be retried.

The gpt-oss-120B model had a tendency to end up in long/infinite cycles during its reasoning process, causing inconsistent and long runtimes. The model also suffered from several anomalies that mainly affected the precision of the model. In some cases, it detected the same antipattern occurrence twice, while in others it denoted an antipattern-free file with a detected antipattern occurrence with a line span consisting of values between -1 and 0, ignoring the instructions in the prompt. In numerous cases, the model filled the output fields with nonsensical values, such as choosing line ranges that extend beyond the bounds of the analysed files, which hinders both the precision and recall of the results in our methodology.

In Figure 12, we display an example where the reasoning process of the model "leaked" into the final response, causing multiple anomalies for one antipattern occurrence: an

invalid line range, an incorrect code fragment containing Unicode-encoded non-breaking space characters, and a reasoning for the antipattern occurrence where the model is berating its previous thinking process. Interestingly, the occurrence of such anomalies was higher for files, which required more reasoning tokens to be analysed. We did not spot any such anomalies in the results for other models. An interpretation of these stability issues is discussed in Section 8.1.1.

```
Response(occurrences=[AntipatternOccurrence(
    antipatternName=<AntipatternName.id_required: 'ID Required'>,
    linesRangeStart=0,
    linesRangeEnd=-17,
    codeFragment='77: \xa0\xa0',
    reasoning="]} The assistant's answer is garbled nonsensical. Need
    to produce correct output: list of antipattern occurrences with
    appropriate fields: antipatternName, codeFragment (lines of
    definition), linesRangeStart, linesRangeEnd, and reasoning. Let's
    analyze class for each antipattern defined: ID Required, Keyless
    Entry, Rounding Errors, 31 Flavors, Fear of the Unknown. Also
    consider that class is for a table (not a view). First, ID Required
    : primary key column name "
)])
```

Figure 12. Example of an anomalous response provided by the gpt-oss-120B model.

7.2 RQ2: How do different prompting strategies affect the detection performance of LLMs in identifying SQL antipatterns in Java code that uses jOOQ for database access?

In the following sections, we analyse the effects of the evaluated prompting strategies on the detection performance of SQL antipatterns. It should be noted that due to the stability issues described in the previous section, which resulted in extremely inconsistent runtimes for the GLM-5 and gpt-oss-120B models, we omitted these models from all runtime comparisons between the evaluated prompting strategies and the baseline.

7.2.1 Few-Shot Prompting

As shown by the binary evaluation metrics in Table 5, only GLM-5 showed a slight improvement in the precision (0.89 vs 0.88), recall (0.90 vs 0.89), and F1-score (0.89 vs 0.88). For all other models, the improvements, if any, were within the margin of error.

Table 5. Binary evaluation metrics, costs, runtimes, and retry counts for detection of SQL antipatterns using Few-Shot Prompting.

Model	Precision	Recall	F1-Score	Cost (USD)	Runtime (s)	Retries
OpenAI GPT-5.2 (reasoning)	0.86	0.92	0.88	29.97	2507	0
Z.ai GLM-5 (reasoning)	0.89	0.90	0.89	14.85	4281	9
Anthropic Claude Opus 4.5 (non-reasoning)	0.88	0.89	0.88	45.00	352	0
OpenAI gpt-oss-120B (reasoning)	0.83	0.87	0.84	2.00	543	1

In Appendix 11 we show the detection performances for individual SQL antipatterns using the Few-Shot Prompting strategy. The differences compared to the baseline were generally negligible, except for the gpt-oss-120B model, which saw the F1-score for the "Keyless Entry" antipattern decrease from 0.77 to 0.71, and the F1-score for the "Fear of the Unknown" antipattern increase from 0.18 to 0.42.

Using the Few-Shot Prompting strategy, the average analysis cost across all models increased by 31 %, showing a significant increase in cost at a negligible increase in detection performance. On the contrary, the runtime did not see significant changes compared to the baseline: for GPT-5.2 it increased by 3 %, while for Opus 4.5 it **decreased** by 4 %. While the runtimes of the two open models decreased significantly, these changes were an effect of the stability issues affecting the models.

7.2.2 Chain-of-Thought Prompting

Using the CoT strategy, the average analysis cost across all models increased by 23 % compared to baseline, while the runtime for GPT-5.2 increased by 20 % and the runtime for Opus 4.5 increased by 160 %, as shown in Table 6.

Table 6. Binary evaluation metrics, costs, runtimes, and retry counts for detection of SQL antipatterns using Chain-of-Thought Prompting.

Model	Precision	Recall	F1-Score	Cost (USD)	Runtime (s)	Retries
OpenAI GPT-5.2 (reasoning)	0.85	0.91	0.87	30.43	2896	1
Z.ai GLM-5 (reasoning)	0.87	0.88	0.87	15.39	9628	234
Anthropic Claude Opus 4.5 (non-reasoning)	0.92	0.90	0.90	40.86	956	0
OpenAI gpt-oss-120B (reasoning)	0.81	0.85	0.82	1.62	895	3

The prompting strategy did not improve performance over the baseline for any of the reasoning models; on the contrary, the F1-score decreased slightly for all of them. The non-reasoning model, Claude Opus 4.5, on the other hand, saw small improvements in the F1-score, increasing from 0.88 to 0.90.

In Appendix 12 we show the detection performances for individual SQL antipatterns using the CoT prompting strategy. Performance improvements in Opus 4.5 can be seen for multiple antipatterns: for the "Implicit Columns" antipattern, both precision and recall metrics improved noticeably (from 0.94 to 0.97, and from 0.93 to 0.95), while the "ID Required" antipattern saw significant improvements in detection precision (from 0.90 to 0.95). Contrarily, the precision for the "Keyless Entry" antipattern dropped from 0.77 to 0.72. The precision for the "Fear of the Unknown" antipattern increased from 0.60 to 0.88, while the recall dropped from 0.50 to 0.39, resulting in the F1-score remaining nearly identical.

While all other models also experienced fluctuations in their precision and recall scores for individual antipatterns, these fluctuations were mostly negligible. The most notable exception was the "31 Flavors" antipattern for the GPT-5.2 model, which saw a significant decrease in both precision and recall, resulting in the F1-score decreasing from 0.75 to 0.58. We observed the largest overall decline in the precision and recall scores of the gpt-oss-120B model, further discussed in Section 8.1.1.

7.2.3 Tree-of-Thought Prompting

The ToT strategy was by far the most computationally expensive: the average analysis cost (across all models) increased by 88 % compared to baseline, while the runtime for GPT-5.2 increased by 94 %, and the runtime for Opus 4.5 increased by a staggering 1181 %, as shown in Table 7.

Table 7. Binary evaluation metrics, costs, runtimes, and retry counts for detection of SQL antipatterns using Tree-of-Thought Prompting.

Model	Precision	Recall	F1-Score	Cost (USD)	Runtime (s)	Retries
OpenAI GPT-5.2 (reasoning)	0.87	0.91	0.89	51.70	4702	2
Z.ai GLM-5 (reasoning)	0.92	0.88	0.87	16.42	5359	14
Anthropic Claude Opus 4.5 (non-reasoning)	0.92	0.90	0.90	81.43	4701	9
OpenAI gpt-oss-120B (reasoning)	0.81	0.83	0.81	2.13	2139	58

The results were similar to the ones of the CoT prompting strategy, with the non-reasoning model, Claude Opus 4.5, seeing the largest improvement in the F1-score, increasing from 0.88 to 0.90. The F1-score dropped slightly for two of the three reasoning models, but interestingly, the GPT-5.2 model saw slight improvements in its detection performance. Once again, the gpt-oss-120B model exhibited the largest overall reduction in detection performance, exceeding even the decline observed with the CoT strategy.

In Appendix 13 we show the detection performances for individual SQL antipatterns using the ToT prompting strategy. Similarly to the CoT strategy, performance improvements in Opus 4.5 can be seen for multiple antipatterns: for the “Implicit Columns” antipattern, both precision and recall metrics improved slightly, while the “ID Required” antipattern saw significant improvements in detection precision (from 0.90 to 0.96). The precision for the “Fear of the Unknown” antipattern increased from 0.60 to 0.87, while the recall dropped from 0.50 to 0.39, resulting in an overall decrease in the F1-score.

While all other models also saw fluctuations to their precision and recall scores for individual antipatterns, they were mostly negligible. The most notable exception was the “Keyless Entry” antipattern for the GPT-5.2 model, which saw a significant increase in precision,

increasing from 0.53 to 0.65.

7.2.4 Summary

None of the evaluated prompting strategies showed a consistent improvement in detection performance across all models. Prompting strategies designed to elicit reasoning (CoT, ToT) displayed a 2 % increase in the F1-score for the non-reasoning model, while often degrading the performance of reasoning models. Few-Shot Prompting increased the F1-score for GLM-5 while not affecting the performance of the other models.

Compounded with the fact that the more complex prompting strategies resulted in a large increase in analysis costs and runtimes, we ultimately found that they were not reliably beneficial in this setting. Therefore, we used our baseline Zero-Shot prompts for the development of our SQL antipattern detection tool.

7.3 RQ3: How accurately can the developed LLM-based tool detect SQL antipatterns in Java code that uses jOOQ for database access?

We performed the analysis of the tool using the Claude Opus 4.5 model, with reasoning disabled, and temperature set to 0.0. The tool performed the analysis using the previously developed Zero-Shot prompts. We selected the Claude Opus 4.5 model for the analysis because it was tied for the best performance under the Zero-Shot strategy while also being by far the fastest model evaluated. The cost of the analysis of 20 projects (containing 502 analysed "relevant" files) was $14.34 and the runtime was 386 seconds.

Importantly, during the subsequent qualitative analysis of the tool's output (discussed in detail in Section 8.1.2), we discovered several inaccuracies and omissions in our human-annotated ground truth. Because these human errors negatively affected the tool's performance metrics, we evaluated the tool against both the original (uncorrected) and revised (corrected) ground truth datasets.

Table 8 summarises the absolute detection counts—TPs, FPs, and FNs—for each individual antipattern when evaluated against the original, uncorrected ground truth. Detailed confusion matrices for each individual antipattern, reflecting both the uncorrected and corrected datasets, are available in Appendix 14.

Table 8. Absolute detection counts of SQL antipatterns using the developed tool (uncorrected).

Antipattern	**TP**	**FP**	**FN**
Implicit Columns	253	4	17
ID Required	89	13	16
Keyless Entry	35	30	1
Fear of the Unknown	19	24	17
31 Flavors	38	1	1
Poor Man's Search Engine	13	4	8
Rounding Errors	13	0	3

In Tables 9 and 10, we show the overall confusion matrix for the tool, before and after applying corrections regarding mistakes in the ground truth.

Table 9. Confusion matrix for detection of SQL antipatterns using the developed tool.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	460	63	523
	Absent	76	N/A[1]	76
	Total	536	63	599

Table 10. Confusion matrix for detection of SQL antipatterns using the developed tool (corrected).

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	512	51	563
	Absent	24	N/A	24
	Total	536	51	587

We calculated the detection performance of our developed tool for each individual antipattern, as well as the overall detection performance, calculated as the weighted average of the detection performances of individual antipatterns. As the corrections to

[1]There is no clear way of calculating the number of True Negatives in the context of antipattern localisation, and the number would dwarf all other numbers in the table without offering much additional value. Therefore, we omit the number of True Negatives from our confusion matrices.

ground truth are inherently biased towards impacting detection performances positively, we provide detection performances for both uncorrected and corrected ground truths, shown in Tables 11 and 12.

Table 11. Binary evaluation metrics for detection of SQL antipatterns using the developed tool (uncorrected).

Antipattern	**Precision**	**Recall**	**F1-Score**
Implicit Columns	0.98	0.94	0.96
ID Required	0.87	0.85	0.86
Keyless Entry	0.54	0.97	0.69
Fear of the Unknown	0.44	0.53	0.48
31 Flavors	0.97	0.97	0.97
Poor Man's Search Engine	0.76	0.62	0.68
Rounding Errors	1.00	0.81	0.90
Total	0.88	0.88	0.88

Table 12. Binary evaluation metrics for detection of SQL antipatterns using the developed tool (corrected).

Antipattern	**Precision**	**Recall**	**F1-Score**
Implicit Columns	1.00	0.94	0.97
ID Required	0.99	0.94	0.96
Keyless Entry	0.75	1.00	0.86
Fear of the Unknown	0.98	0.71	0.82
31 Flavors	0.97	0.97	0.97
Poor Man's Search Engine	0.76	0.62	0.68
Rounding Errors	1.00	0.81	0.90
Total	0.96	0.91	0.93

7.3.1 Without Localisation

We also performed the analysis using the classification mode, which detects antipatterns on a per-file basis, without specifying the exact location. The other settings remained the same: we used the Claude Opus 4.5 model, with reasoning disabled, and temperature set to 0.0. With these settings, the cost of the analysis was $12.24 and the runtime was 295 seconds.

Similarly to before, we analysed the FPs and FNs in the results, and made corrections to errors found to be mistakes in the ground truth. In Appendix 15, we show the confusion matrices for each individual antipattern, along with corrected versions where necessary.

In Tables 13 and 14, we show the overall confusion matrix for the tool, before and after applying corrections regarding mistakes in the ground truth.

Table 13. Confusion matrix for detection of SQL antipatterns using the developed tool without localisation.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	244	20	264
	Absent	63	N/A	63
	Total	307	20	327

Table 14. Confusion matrix for detection of SQL antipatterns using the developed tool without localisation (corrected).

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	283	14	297
	Absent	24	N/A	24
	Total	307	14	321

Based on the uncorrected and corrected confusion matrices, we calculated the detection performance of our developed tool for each individual antipattern, as well as the overall detection performance, calculated as the weighted average of the detection performances of individual antipatterns, shown in Tables 15 and 16.

Table 15. Binary evaluation metrics for detection of SQL antipatterns using the developed tool without localisation (uncorrected).

Antipattern	**Precision**	**Recall**	**F1-Score**
Implicit Columns	0.95	0.99	0.97
ID Required	0.95	0.90	0.93
Keyless Entry	0.41	1.00	0.58
Fear of the Unknown	0.35	0.88	0.50
31 Flavors	1.00	1.00	1.00
Poor Man's Search Engine	1.00	0.91	0.95
Rounding Errors	1.00	0.60	0.75

Continues...

Table 15 – *Continues...*

Antipattern	**Precision**	**Recall**	**F1-Score**
Total	0.88	0.92	0.88

Table 16. Binary evaluation metrics for detection of SQL antipatterns using the developed tool without localisation (corrected).

Antipattern	**Precision**	**Recall**	**F1-Score**
Implicit Columns	0.97	1.00	0.98
ID Required	1.00	0.95	0.98
Keyless Entry	0.77	1.00	0.87
Fear of the Unknown	0.74	0.94	0.83
31 Flavors	1.00	1.00	1.00
Poor Man's Search Engine	1.00	0.91	0.95
Rounding Errors	1.00	0.60	0.75
Total	0.93	0.95	0.94

As seen from Tables 15 and 16, the weighted average detection performance remained similar to the localised version. Interestingly, the tool became significantly more paranoid in detecting several antipatterns ("Implicit Columns", "Fear of the Unknown"), displaying a lower precision at a higher or similar recall. The "Poor Man's Search Engine" antipattern, which suffered from issues with localisation behaviour (discussed in Section 8.1.2), saw a significant increase in both detection precision (1.00 vs 0.76) and recall (0.91 vs 0.62).

Overall, the tool shows a strong detection performance for most evaluated SQL antipatterns, both in localisation and classification modes. The most difficult antipatterns to detect, both for LLMs and human annotators, are "Keyless Entry" and "Fear of the Unknown", which require the most thorough understanding of the relationships of entities and business logic in the codebase.

7.4 RQ4: What patterns can be observed in the occurrence of SQL antipatterns in Java code that uses jOOQ for database access?

We analysed the complete set of 602 projects (containing 17 450 analysed "relevant" files) with our tool, using the Claude Opus 4.5 model, with reasoning disabled, and temperature set to 0.0. The cost of the analysis was $1036.32 and the runtime was 12 391 seconds (3h

31m). The results of the analysis are shown in Table 17.

Table 17. Prevalence of SQL antipatterns in analysed projects.

Antipattern	**Occurrence Count**	**Containing Project Count**	**Occurrences per Project**	**Occurrences per Containing Project**	**Occurrences per 100 jOOQ Statements**
Implicit Columns	7289	535	12.11	13.62	21.94
ID Required	3597	523	5.98	6.88	10.83
Keyless Entry	2153	200	3.58	10.77	6.48
Fear of the Unknown	1256	205	2.09	6.13	3.78
31 Flavors	390	87	0.65	4.48	1.17
Poor Man's Search Engine	583	114	0.97	5.11	1.75
Rounding Errors	663	86	1.10	7.71	2.00
Total	15 931	601	26.46	26.51	47.96

In total, we flagged 15 931 occurrences of the seven detected antipatterns. "Implicit Columns" and "ID Required" are by far the most prevalent antipatterns, existing in 89 % and 87 % of the analysed projects, respectively. Although the remaining antipatterns appear in far fewer projects—with "Fear of the Unknown" leading the secondary group at 34 %—their concentration within affected projects remains high.

By evaluating Jaccard Indexes, Conditional Probabilities, and Spearman correlations (detailed in Appendices 16–21), we identified distinct co-occurrence patterns among these antipatterns:

- **The Ubiquitous Pair:** "Implicit Columns" and "ID Required" frequently co-exist (Jaccard: 0.81) and scale together in intensity (Spearman: 0.43).
- **Constraint Neglect:** "Keyless Entry" and "Fear of the Unknown" demonstrate a mutual linkage. Half of the projects containing one will also contain the other,

sharing a moderate positive correlation (Spearman: 0.32).

- **Query Decay vs Schema Decay:** At the file level, design and query antipatterns exhibit negative correlations, naturally segregating into different files. Conversely, query antipatterns like "Poor Man's Search Engine" and "Implicit Columns" often share the same files.

Furthermore, the conditional probability data highlights that the presence of rare antipatterns acts as a weak indicator for common ones. The deeper implications of these co-occurrence patterns are explored further in Section 8.1.4.

7.5 RQ5: Which jOOQ API methods are most frequently associated with query antipattern occurrences?

In the following sections, we examine the jOOQ API methods most commonly associated with the "Implicit Columns" and "Poor Man's Search Engine" SQL antipatterns.

7.5.1 Implicit Columns

As shown in Table 18, we found that the most frequently associated methods for the "Implicit Columns" antipattern are `DSL.selectFrom(TABLE)` and `DSL.select().from(TABLE)`, making up nearly three quarters of all "Implicit Columns" antipattern occurrences in total.

Calls to `TABLE.fields()` and `DSL.asterisk()` are associated with 9.1 % and 3.8 % of all occurrences. Generated DAO methods and plain SQL queries containing $*$ wildcards are both associated with close to 2 % of all occurrences.

Table 18. jOOQ API methods most commonly associated with the "Implicit Columns" SQL antipattern.

Method Signature	**Antipattern Occurrence Count**	**Percentage of All Occurrences**
`DSL/DSLContext.selectFrom(TABLE)`	3417	46.9
`DSL/DSLContext.select().from(TABLE)`	1810	24.8
`TABLE.fields()`	665	9.1
`DSL.asterisk()`	280	3.8
Generated DAO methods (combined)	163	2.2
`DSL/DSLContext.fetch(TABLE, ...)`	151	2.1
Plain SQL wildcards	137	1.9

Continues...

Table 18 – *Continues...*

Method Signature	**Antipattern Occurrence Count**	**Percentage of All Occurrences**
`DSL/DSLContext.fetchOne(TABLE, ...)`	130	1.8
`DSL/DSLContext.select(TABLE).from(TABLE)`	107	1.5
Other/uncategorised	429	5.9

7.5.2 Poor Man's Search Engine

As shown in Table 19, the method most commonly associated with the "Poor Man's Search Engine" SQL antipattern is the `Field.like` DSL method with embedded `%` wildcards, accounting for nearly half of all antipattern occurrences at 46.8 %. The case-insensitive variant, `Field.likeIgnoreCase`, makes up another 14.9 % of all occurrences.

DSL methods `Field.containsIgnoreCase` and `Field.contains` are associated with 16.5 % and 10.5 % of all occurrences. Plain SQL queries with the `LIKE` operator and preceding and succeeding wildcards make up 2.9 % of all occurrences, while regex matching operations using the `Field.likeRegex` DSL method make up 2.1 % of all occurrences.

Table 19. jOOQ API methods most commonly associated with the "Poor Man's Search Engine" SQL antipattern.

Method Signature	**Antipattern Occurrence Count**	**Percentage of All Occurrences**
`Field.like`	273	46.8
`Field.containsIgnoreCase`	96	16.5
`Field.likeIgnoreCase`	87	14.9
`Field.contains`	61	10.5
Plain SQL `LIKE`	17	2.9
`Field.likeRegex`	12	2.1
Other/uncategorised	37	6.3

8 Analysis

In this chapter, we critically analyse the empirical findings of our study to understand the capabilities and limitations of LLMs in detecting the seven evaluated SQL antipatterns. We also reflect on the successes and shortcomings of the research process, discuss the study's limitations, and outline directions for future research.

8.1 Interpretation of Results

This section discusses the results of our quantitative and qualitative analysis, exploring the underlying causes of model behaviour, comparing our findings to existing literature, and examining the co-occurrence patterns of SQL antipatterns.

8.1.1 LLM Capabilities and Limitations in Static Analysis

Across all prompting strategies, we noticed a pattern where more common antipatterns were generally detected with a higher performance than rarer antipatterns. This can have multiple contributing factors:

- **Lack of training data:** Due to fewer cases in the training set, the instructions for detecting less common antipatterns may not be as refined and resilient to variations in coding patterns.
- **Low confidence:** Due to fewer cases in the validation set, each individual detection error causes a large reduction of the F1-score for less common antipatterns.
- **Detection complexity:** The most common antipatterns, "Implicit Columns" and "ID Required", are in most cases self-contained and do not require cross-referencing multiple pieces of code, while some less common antipatterns, such as "Keyless Entry" and "Fear of the Unknown" require additional context to be detected correctly.

When evaluating the Few-Shot Prompting strategy, the improvements we observed during development did not materialise in our evaluations on the validation set. We suspect that

this is due to a classic case of overtraining: the provided examples were largely based on the training set, which guided the models to make the correct decisions for the edge cases present in the training set, but they did not have a large impact when tested on a different data set. The only exception was the gpt-oss-120B model, indicating that the additional clarity from the provided examples had an outsized impact for the gpt-oss-120B model compared to others.

Regarding the CoT strategy, the large increases in runtime we observed indicate that the prompt functions as expected by eliciting additional reasoning in the models. The increases in runtime align with the results of Meincke et al. [55, p. 3–6], who found CoT prompts to take anywhere from 20–80 % more time than direct prompts for reasoning models, and 35–600 % for non-reasoning models.

In terms of detection performance, our results also loosely align with the findings of Meincke et al. [55, p. 3–6], who found the effect of CoT prompting to be nuanced for both reasoning and non-reasoning models. Although they found the average performance of non-reasoning models to generally improve (in the range of 4.4 % and 13.5 %), CoT introduced additional variability, causing errors that would have not occurred without it. For reasoning models, it only yielded marginal improvements on average, while also negatively affecting the performance of some models. It should be noted that the models used in their research were generally multiple generations older, architecturally different, and smaller than the models used by us, which could explain the discrepancies between our results.

During the evaluation, the gpt-oss-120B model had a tendency to end up in long or even infinite cycles during its reasoning process, causing inconsistent and long runtimes. The model also frequently produced anomalous responses, first discussed in Section 7.1, which hindered both the precision and recall of the model. We observed that the anomalies were more prevalent for files, which required more reasoning tokens to be analysed.

When evaluating both the CoT and ToT strategies, we also observed the largest overall decline in the precision and recall scores of the gpt-oss-120B model, along with an increased prevalence of anomalies. As CoT and ToT were the most reasoning-heavy prompting strategies, these observations are consistent with the amplification of “context rot”—a

phenomenon where LLM performance consistently degrades and becomes increasingly unreliable as input context length grows [109], but this causal explanation was not isolated experimentally.

8.1.2 Qualitative Analysis of Detection Errors

In this section, we qualitatively analyse the underlying causes of the FPs and FNs observed during the evaluation of the developed tool (as presented in Section 7.3). We structure this analysis by the respective SQL antipatterns.

Implicit Columns

The main causes of FNs were DML statements, which returned all columns of modified rows with the `returning()` and `returningResult()` methods, which translate to the `RETURNING *` clause in plain SQL. It appears that the model did not associate these methods with the "Implicit Columns" antipattern, although they return blind projections from the database. Such DML statements accounted for 14 of the 17 FNs.

One FP and one FN were caused by different localisation behaviour: while the ground truth annotated the entire multi-line `SELECT` clause, the tool only attributed the antipattern to the line directly causing the antipattern.

The final three FPs and two FNs were caused by mistakes in the ground truth: in one case, an antipattern occurrence was attributed to the wrong source file; in the second case, the line number was off by one; in the final case, the ground truth was missing an annotation altogether.

ID Required

Six FNs were caused by the tool not flagging primary key columns named as "id", while one FN was caused by the tool not recognising an opportunity to use an alternative ID field as the primary key instead of the synthetic primary key. One FP was caused by the tool suggesting replacing a synthetic primary key with a field, which would make foreign keys too large and complex (a field containing internet URLs).

The remaining FPs and FNs were caused by mistakes in the ground truth: in five cases, the ground truth contained incorrect line ranges for antipattern occurrences; in four cases, the ground truth contained invalid antipattern occurrences; in seven cases, the ground truth

was missing valid antipattern occurrences detected by the tool.

Keyless Entry

Half of the FPs (15) were caused by the tool flagging missing foreign keys in tables used for audit logging. We did not flag these columns as antipattern occurrences in the ground truth because these tables store historical data, which could refer to rows that do not exist any more. The prompt instructions did not specify an exception for audit tables; therefore, these errors were likely caused by imprecise instructions.

One FP was caused by an ambiguity in database object names. The affected schema contained a table named "island", which had a primary key column named "id". The second table contained a column named "island_id", which the tool considered a reference to the aforementioned primary key. However, the columns had different data types, and "island_id" actually referred to an external identifier.

The other 14 FPs and one FN were caused by mistakes in the ground truth: in one case, the ground truth contained an incorrect line range for an antipattern occurrence; in the other 13 cases, the ground truth was missing valid antipattern occurrences.

Nine of these FPs were edge cases, which we decided to resolve in favour of the tool: the tool flagged nine columns storing usernames of application users. While the affected database schema contained a table for users, the username was neither a primary nor unique key in that table. We considered the missing unique key an integrity violation of its own, and resolving it would have resulted in definite cases of the "Keyless Entry" antipattern; hence, we considered these nine detections to be mistakes in the ground truth.

Fear of the Unknown

Ten FNs were caused by the tool not flagging columns, which used `NOT NULL` constraints with special default values (e.g., an empty string or the Unix epoch (January 1, 1970, 00:00:00 UTC)) to indicate "missing" values, rather than using nullable columns.

The other seven FNs were caused by the tool not flagging columns, which had default values and contained values that were non-nullable by nature (e.g., boolean values indicating state).

One FP was caused by the model suggesting a refactoring opportunity in a complex query,

rather than pointing out an actual antipattern occurrence.

The other 23 FPs were all caused by mistakes in the ground truth, which was missing many "Fear of the Unknown" antipattern occurrences. In six cases, it was missing columns, which used default values to represent missing values; in the other 17 cases, it was missing columns, which were nullable, but contained values non-nullable by nature.

31 Flavors

The single FN was caused by the tool not flagging the `CHECK` constraint shown in Figure 13, which validated a `SMALLINT` column against a range of values representing different authorisation methods. This error was likely due to imprecise prompt instructions because the prompt contained an instruction to ignore `CHECK` constraints checking for ranges (e.g., valid ages), but did not account for edge cases of this kind.

```
(((authorize_type >= 0) AND (authorize_type <= 1)))
```

Figure 13. CHECK constraint causing a false negative "31 Flavors" detection.

The single FP was caused by the tool flagging a `VARCHAR(50)` column named "status", while there was nothing in the schema definition code indicating that the column contained a fixed set of values.

Poor Man's Search Engine

Seven FNs and two FPs were caused by differences in localisation behaviour. In cases where two or more consecutive lines contained distinct occurrences of the "Poor Man's Search Engine" antipattern, the ground truth contained annotations for each of them separately. The tool grouped them together as one antipattern occurrence, although the prompt contained instructions to do otherwise.

The remaining two FPs were caused by the tool flagging usages of the `LIKE` operator without any wildcards as antipattern occurrences. In these cases, the `LIKE` operator behaved similarly to an ordinary equality check, which could be determined from the logic of the class being analysed.

The final FN was caused by the tool not flagging a usage of the `containsIgnoreCase` jOOQ DSL method, which is translated into the `LIKE` operator with preceding and succeeding wildcards. We suspect that the antipattern occurrence was not flagged because

the condition was stored in an intermediate Java collection before being used in a query.

Rounding Errors

The three FNs were all the same kind: the model did not consider casino game coefficients stored in `DOUBLE` columns as "Rounding Errors" antipattern occurrences. However, we annotated these columns as antipattern occurrences in the ground truth because these coefficients were used to calculate monetary winning in the business logic.

8.1.3 Comparison with Existing Tools and Literature

Sousa et al. [48] were able to classify a variation of the "Implicit Columns" antipattern (called the "Use of `SELECT *` (asterisk wildcard)" or "USA" issue in their work) in PL/SQL projects with an F1-score of 1.00 using the GPT-4o model, with Phi-4, Gemini 2.0 Flash, and GPT-4o mini models displaying a lower F1-score. While achieving a higher detection performance than us, their work only included a subset of the "Implicit Columns" antipattern, simplifying the detection task for models. In total, they detected eight PL/SQL code smells, with the GPT-4o and Phi-4 models achieving an overall F1-score of 0.79.

Nagy and Cleve [60] did not measure the detection performance of their tool, *SQLInspect*. However, their tool relies on SQL query extraction from Java code. In their performance evaluation against multiple codebases, the query extractor achieved a precision of at least 0.879 and a recall of at least 0.715 [110, p. 492], resulting in a minimum F1-score of 0.79 for SQL query extraction, which they suggest using as an upper bound for the SQL antipattern detection performance of *SQLInspect* [4, p. 328]. The overlap of detected antipatterns between our tool and *SQLInspect* is limited to the "Fear of the Unknown" and "Implicit Columns" antipatterns.

The suggested upper bound F1-score of 0.79 is lower than both the corrected (0.97) and uncorrected (0.96) F1-scores of our tool for the "Implicit Columns" antipattern. It is also lower than the corrected F1-score of our tool for the "Fear of the Unknown" antipattern (0.82), but higher than the uncorrected F1-score (0.48). However, it should be noted that *SQLInspect* only detects the "Fear of the Unknown" antipattern in DQL and DML statements, and does not detect schema-level issues.

In Table 20, we show the detection performance of our tool in classification mode, compared

against works regarding LLM-based antipattern, smell, and vulnerability detection in various domains. Based on F1-score, we selected the results of the best-performing model(s) from each work. While not a direct comparison, it shows that our tool performs adequately compared to other tools using a similar methodology in other domains.

Table 20. Detection performance of the developed tool compared to various LLM-based detection tools.

Tool	**Year**	**Model**	**Domain**	**Precision**	**Recall**	**F1-Score**
Tamberg and Bahsi [111]	2024	GPT-4	Software vulnerability detection	0.76	0.60	0.67
Sousa et al. [48]	2025	GPT-4o	PL/SQL smell detection	0.68	0.93	0.79
Sousa et al. [48]	2025	Phi-4	PL/SQL smell detection	0.70	0.90	0.79
Blaauwendraad [46]	2025	GPT 4o-mini	Traditional code smell detection	0.26	0.61	0.37
Sadik and Govind [47]	2025	GPT-4	Traditional code smell detection	0.79	0.41	0.54
Ploom and Eessaar [112]	2026	Gemini 3 Pro	UML diagram smell detection from PDFs	0.82	0.69	0.75
Ours (uncorrected)	2026	Claude Opus 4.5 (non-reasoning)	SQL antipattern detection in jOOQ	0.88	0.92	0.88
Ours (corrected)	2026	Claude Opus 4.5 (non-reasoning)	SQL antipattern detection in jOOQ	0.93	0.95	0.94

Compared to Muse et al. [4, p. 322], who found the “Implicit Columns” antipattern to be present in 1.67 % of all SQL statements, and the “Fear of the Unknown” antipattern to be present in 0.8 % of all SQL statements, our prevalence numbers are vastly larger.

While our metric—occurrences per 100 jOOQ statements—allows for multiple occurrences within a single statement, Muse et al. used a binary count for each antipattern, recording only its presence or absence per statement. Due to this difference in granularity, as well the difference in statement count estimations (explained in Section 6.1), these numbers are not directly comparable. Despite this, they can be used as indications of broader trends.

We found there to be 21.94 occurrences of “Implicit Columns” per 100 SQL statements, which is over 13 times more than found in SQL statements by Muse et al. [4, p. 322]. We

believe that this difference originates from a difference in detection methodologies: while *SQLInspect*, which was used by Muse et al., only detects wildcards in `SELECT` statements [60, p. 150], we also detect other blind projections generated by jOOQ, which comprised the vast majority of "Implicit Columns" antipattern occurrences detected by us (as shown in Section 8.1.5).

We also found there to be 3.78 occurrences of "Fear of the Unknown" per 100 SQL statements, which is nearly five times more than found in SQL statements by Muse et al. [4, p. 322]. We believe that this difference also originates from a difference in detection methodologies: while *SQLInspect* detects the antipattern only from DML and DQL statements [60, pp. 149–150], we also detect it from the database schema. As seen from the file-level Jaccard Indices (Appendix 17), we found the majority of "Fear of the Unknown" antipattern occurrences from classes representing the database schema, alongside design antipatterns.

8.1.4 The Anatomy of SQL Decay

We found the "Implicit Columns" and "ID Required" antipatterns to be almost ubiquitous across the dataset. Because they are present in 89 % and 87 % of projects respectively, their high project-wise Jaccard Index (0.81) is largely a byproduct of their high individual prevalence. Nevertheless, their near-ubiquity suggests that they represent baseline characteristics of poor SQL design in the analysed projects. Furthermore, their moderate project-wise Spearman correlation (0.43) suggests that these antipatterns tend to increase together: projects with more generic primary keys often also contain more queries with blind projections.

The "Keyless Entry" and "Fear of the Unknown" antipatterns form a second, lower-prevalence co-occurrence pattern. They are the third and fourth most prevalent antipatterns, appearing in 33 % and 34 % of projects, respectively. Their project-wise Jaccard Index of 0.34 indicates that roughly one third of projects exhibiting at least one of the two antipatterns exhibit both. This association is also reflected in the conditional probabilities: approximately half of the projects with "Keyless Entry" also exhibit "Fear of the Unknown", and vice versa. Together with their moderate project-wise Spearman correlation of 0.32, these results suggest a possible broader tendency towards underutilising database-level

constraints: projects omitting `NOT NULL` constraints are also more likely than average to omit foreign key constraints, and vice versa, likely shifting data integrity responsibilities to the application layer.

As expected, the file-level Spearman correlations between DDL and DML/DQL antipatterns are negative, since these antipatterns generally cannot occur within the same source files. The positive file-level Spearman correlation between "Implicit Columns" and "Poor Man's Search Engine" indicates that while most antipatterns are isolated at the file level, these query antipatterns show a slight localised tendency to increase together.

The Conditional Probability analysis indicates that rarer SQL antipatterns frequently co-occur with more prevalent ones, rather than appearing in isolation. For instance, finding the "Poor Man's Search Engine" antipattern yields a 95 % probability of also finding the "Implicit Columns" antipattern within the same project. Given that "Implicit Columns" already has a high baseline prevalence (present in 89 % of all projects), this high conditional probability suggests that niche antipatterns rarely exist in isolation; a codebase exhibiting rarer, specific antipatterns almost universally suffers from more common antipatterns as well.

8.1.5 Associations with jOOQ API Design

Our analysis of the jOOQ API methods associated with SQL antipattern occurrences highlights several implications for the library's documentation and possible API design review.

Regarding the "Implicit Columns" antipattern, we found that jOOQ DSL methods, which fetch records containing all columns from the underlying database table(s), make up nearly three quarters of all occurrences. Although the jOOQ documentation explicitly discourages the use of such methods in order to avoid fetching redundant data [81], they represent some of the simplest and most straightforward approaches to data retrieval in jOOQ and appear frequently among the detected antipattern occurrences. In contrast, calls to methods, which make the intent of fetching all columns from tables explicit, are much less frequently associated with the antipattern.

For the "Poor Man's Search Engine" SQL antipattern, while the `Field.like` DSL method is

directly translated into an equivalent SQL statement, a significant portion of occurrences are associated with DSL methods like `Field.containsIgnoreCase` and `Field.contains`. These methods implicitly add preceding and succeeding `%` wildcards to the generated SQL. By hiding the wildcard mechanics, these abstraction methods may make occurrences of the "Poor Man's Search Engine" antipattern less obvious to developers.

8.2 Reflection on Work Process

We performed the reflection of the thesis based on the experiential learning theory by Kolb [113], according to which a concrete experience (performing the work) is followed by reflection, learning from the experience gained, and testing the results in new conditions. In the present work, we did not perform testing in new conditions; however, we highlight what others undertaking similar work could learn from this study.

8.2.1 Successes

The primary success of this research lies in the successful design and implementation of the LLM-powered static analysis CLI tool, as well as the large-scale empirical analysis of 602 open-source jOOQ projects. By formulating antipattern detection as a multi-label, multi-occurrence localisation task, we demonstrated the potential of LLMs to address the parsing limitations of dynamic, DSL-based database frameworks. Furthermore, processing over 17 000 relevant source files allowed us to extract actionable, real-world insights, successfully bridging the gap between theoretical database design flaws and actual developer behaviour in modern Java environments.

Although the final focus of the thesis was settled upon relatively late in the research process, pivoting to an LLM-based methodology ultimately proved to be a critical success. Attempting to achieve similar detection capabilities using traditional static analysis would have been exponentially more complex, partially due to unforeseen complexities. For instance, detecting the "31 Flavors" antipattern requires evaluating fragments of raw SQL `CHECK` constraints embedded as strings within jOOQ-generated Java classes, effectively forcing a fragile, multi-language parsing pipeline. Furthermore, the highly dynamic nature of the jOOQ DSL would have demanded complex, intra-procedural data flow analysis. By leveraging the semantic and contextual understanding of LLMs, we successfully bypassed

these rigid parsing barriers, validating that the modern AI-driven approach was the right tool for the job.

Beyond the technical achievements, the work process itself was a highly successful experiment in human-AI collaboration. The use of a semi-autonomous agentic workflow proved invaluable not only for coding—allowing for rapid iteration of the tool's architecture, concurrency handling, and file-system traversal logic—but also for agentic writing. Leveraging LLMs as writing assistants helped streamline the drafting and structuring of complex arguments, allowing the research effort to remain focused on core complexities, rather than getting bogged down in boilerplate implementation or writer's block. Additionally, undertaking this thesis served as an excellent vehicle for personal and professional growth, significantly strengthening my academic writing skills and providing a great, hands-on opportunity to master LaTeX.

Finally, a major highlight of this research process was the highly pleasant and productive collaboration with my supervisor. His continuous feedback and deep domain expertise were instrumental in shaping the direction of the study. Through our discussions, he helped to level up the overall methodology of the thesis by a significant margin.

8.2.2 Shortcomings and Problems

A significant overarching problem during this research was the severely compressed timeline. The final topic of the thesis only became fully clear near New Year's Eve, which meant that a good amount of previous exploratory work had to be left aside. This pivot drastically shortened the window available for executing the core experiments. Because the scope of the thesis remained remarkably broad despite this temporal constraint, certain aspects of the methodology and analysis may not have received the thorough attention they ultimately deserved.

Consequently, the rushed timeline led to suboptimal data management practices: specifically, we did not adequately preserve intermediate results at each step. The failure to persist raw detection results for individual model evaluations resulted in redundant API calls whenever minor pipeline adjustments were introduced. This oversight not only inflated computational costs but also reduced the overall transparency and reproducibility of the study. Furthermore, this prevented us from rigorously evaluating the statistical significance

of the performance differences between the alternative prompting strategies and the baseline.

Another significant shortcoming of this research was the reliance on a single human annotator for establishing the ground truth dataset. Despite our rigorous efforts to maintain internal consistency—including the creation of decision trees and a 30-day washout period that yielded a high Cohen's Kappa of 0.834—the single annotator setup was ultimately not reliable enough. Qualitative analysis during the evaluation phase revealed numerous human inaccuracies, missed occurrences, and misclassified edge cases. Relying on one individual's subjective interpretation inherently introduced blind spots and bias into the dataset, meaning the LLMs were benchmarked against a fallible human baseline rather than an objective consensus.

Additionally, the research suffered from unfortunate timing regarding the release cycle of new language models. The landscape of LLMs evolves at a rapid pace, and just as we moved past the model evaluation stage, highly compelling small language models were released. Most notably, Google Gemma 4 and Alibaba Cloud Qwen 3.5 and 3.6 became available. Because our evaluation framework was already committed to the selected models, we were unable to test these newly released model families, which otherwise looked to be perfect candidates for high-performance, cost-effective, and locally hosted static analysis.

8.2.3 Suggestions for Future Improvements

Should this study be repeated or expanded upon, it is imperative that a multi-annotator setup be employed for dataset creation. Engaging multiple independent domain experts to review the jOOQ source code and cross-validate antipattern occurrences would dramatically improve the objective correctness of the ground truth. Resolving annotator disagreements through consensus discussions would eliminate the subjective biases and blind spots that hindered the dataset in this study, resulting in a much more resilient benchmark for evaluating LLM performance.

To improve methodological rigour, future iterations of this work must incorporate strict data versioning and artifact tracking practices. Every intermediate result—such as the raw JSON responses returned by the LLMs for each prompt variation—should be persistently stored and logged. Doing so will eliminate the need for redundant API calls, save on research costs, and ensure total transparency, allowing peer researchers to inspect the exact

reasoning steps and outputs generated at any point during the evaluation phase.

8.3 Limitations

A primary limitation of this study is the reliance on a single human annotator for establishing the ground truth dataset. Although an intra-annotator agreement check was conducted after a 30-day washout period, yielding a high Cohen's Kappa of 0.834, this metric only demonstrates internal consistency rather than objective correctness. As revealed during the qualitative analysis of detection errors, the ground truth contained several human inaccuracies, including missed antipattern occurrences, imprecise line ranges, and misclassified edge cases. Subjective interpretations of complex code structures or systematic blind spots regarding specific jOOQ patterns remain unchecked by independent domain experts, meaning the evaluated LLMs are benchmarked against one individual's fallible interpretation rather than an independently verified consensus.

Consequently, the retroactive corrections made to the ground truth during the evaluation phase introduce a bias in favour of the developed tool. While the corrected F1-scores account for human annotation errors successfully identified by the tool (i.e., false positives that were actually true positives), this methodology inherently ignores 'unknown unknowns'—antipattern occurrences missed by both the human annotator and the language model. Because the ground truth was only corrected in locations where the tool flagged an anomaly, the revised performance metrics effectively represent an optimistic upper bound rather than an absolute measure of the tool's detection capabilities.

Another limitation relates to the calculation of antipattern prevalence metrics, specifically occurrences per 100 statements. Due to the dynamic nature of the jOOQ DSL, extracting and counting exact SQL statements statically is highly error-prone. Consequently, jOOQ API method calls were used as a proxy to estimate the total number of executed statements. Because loop constructs and conditional query building can generate a vastly different number of actual SQL statements at runtime than static method references imply, this proxy measurement introduces a margin of error, invalidating direct comparisons with density metrics from previous plain SQL studies.

Regarding external validity, the dataset utilised for evaluating antipattern prevalence consists

entirely of open-source projects mined from GitHub. Open-source repositories often exhibit different architectural patterns, code review cadences, and developer demographics compared to closed-source enterprise systems. Because jOOQ is widely utilised in enterprise environments dealing with complex databases, the prevalence rates and co-occurrence clusters identified in this study may not perfectly generalise to proprietary industrial codebases.

The repository mining process was also constrained by the technical limitations of the GitHub Code Search API, specifically its hard limit of 1000 results per query, which introduced potential blind spots, further explored in Section 3.1.1.

Additionally, the sampling process was not based on a perfectly filtered initial dataset. As the research progressed, it became evident that the dataset contained minor imperfections, such as the inclusion of a duplicated project and an incomplete heuristic list for excluding irrelevant files. Although further corrections were made later in the study, these retroactive adjustments imply that the initial sampling pool used to create the training, validation, and test splits contained slight flaws that marginally skewed the representativeness of the evaluated sample.

Another limitation concerns the evaluation of the prompting strategies, specifically the risk of overtraining observed in the Few-Shot strategy. The examples provided to the models within the Few-Shot prompts were largely derived from the training set. While this effectively guided the models to make correct decisions for specific edge cases during prompt development, these improvements largely failed to materialise when evaluated against the unseen validation set. This discrepancy suggests that the models may have overfitted to the provided examples, limiting their generalisation capabilities when presented with novel code structures.

An additional limitation regarding the evaluation of prompting strategies is the inability to formally determine the statistical significance of the observed performance differences. Consequently, the comparisons between prompting strategies rely solely on absolute metric differences, lacking the rigorous statistical validation required to prove whether the marginal fluctuations in F1-scores were true improvements or merely the result of random variance.

Furthermore, the selection of prompting strategies evaluated in this research was deliberately

restricted to keep the scope of the thesis manageable. The experiments were strictly limited to techniques that could be fully executed within a single request-response cycle, namely Zero-Shot, Few-Shot, and single-prompt variations of Chain-of-Thought and Tree-of-Thought. More complex, multi-agent, or iterative multi-turn prompting frameworks were excluded from this comparative analysis.

From a Design Science Research perspective, the developed CLI artefact was evaluated purely algorithmically (an artificial ex-post evaluation) against an annotated dataset. While the tool achieved a high F1-score, algorithmic performance does not inherently translate to practical utility. Due to time constraints, the study does not evaluate the tool in a naturalistic setting, leaving questions unanswered regarding whether the LLM-generated explanations are easily understood by developers, if the false-positive rate induces alert fatigue, or if the execution times are viable for integration into real-world developer workflows.

Finally, the inherent non-determinism of LLMs poses a fundamental constraint on the tool's reliability. Although we maximised determinism through multiple technical means—specifically by disabling model reasoning (in the final tool), utilising a zero-temperature setting, and enforcing strict structured outputs—LLMs remain fundamentally probabilistic by nature. Consequently, the tool cannot guarantee the absolute, repeatable determinism that developers traditionally expect from static analysis utilities. Furthermore, due to the high financial costs associated with processing large codebases through commercial LLM APIs, the quantitative evaluations in this study were only executed once. Relying on a single evaluation run means the potential variance in detection performance across multiple iterations of the same dataset remains unmeasured.

The use of OpenRouter as an API gateway introduces an additional layer of potential non-determinism affecting future reproducibility. Because OpenRouter dynamically routes requests for open-weights models (such as GLM-5 or gpt-oss-120B) to different backend hosting providers, variations in backend inference engines or quantization methods could introduce token-level output differences. Furthermore, if a provider silently updates a model version behind the same endpoint, future replications of this study could yield different results. Finally, while API caching did not affect our single-pass methodology (as noted in Section 4.3), future researchers attempting to measure detection variance across multiple iterations of the same dataset must explicitly account for (or disable) OpenRouter's

caching semantics to avoid artificially skewed stability metrics.

8.4 Future Work

The catalogue of SQL antipatterns evaluated in this thesis is limited to Karwin's foundational work. Future work could expand this tool to allow for detecting antipatterns from more extensive and specialised taxonomies [14], [19], [20], [21] and Karwin's follow-up book [23], as well as vendor-specific literature [22].

This thesis deliberately restricted the evaluated prompting strategies to single request-response cycles to manage scope. Exploring more complex (e.g., multi-turn or multi-agent) prompting frameworks could offer sophisticated pathways for improvement, refining reasoning pathways and achieving a higher degree of analytical rigour.

The accurate detection of complex antipatterns (e.g., "Keyless Entry" and "Fear of the Unknown") is currently constrained by the limited cross-file context provided in prompts. Implementing a Retrieval-Augmented Generation (RAG) architecture represents a promising methodological enhancement. By indexing the relevant parts of a codebase, a RAG-enabled system could dynamically supply the LLM with comprehensive context, thereby substantially improving both precision and recall for context-dependent antipatterns. To offset the token overhead of retrieving this additional context, future implementations could pair RAG with lightweight pre-filtering mechanisms, such as regular expressions or basic AST parsing. By using these heuristics to isolate and send only the specific code blocks containing jOOQ queries—rather than entire source files—the tool could drastically shrink the required context windows, conserving API resources and significantly reducing analysis costs.

The financial costs and latency associated with utilising frontier commercial models present a significant barrier to the continuous analysis of large-scale codebases. The landscape of LLMs is evolving rapidly, and during the later stages of this study, several promising families of small language models (Alibaba Cloud Qwen 3.5 and 3.6, Google Gemma 4) were released, potentially offering significant improvements over the medium-sized model (gpt-oss-120B) evaluated in this study. Future research could also include small language models, and explore fine-tuning them using the manually annotated ground truth

dataset developed in this study. These approaches could achieve comparable detection performance while drastically reducing costs. Such models are also viable for self-hosting, ensuring data privacy for proprietary enterprise applications.

The current tool implementation only provides core detection functionality, which future work could extend upon. The architecture allows alternative user interfaces to be developed independently of the tool, while more interactive interfaces could enable more advanced functionality. For instance, a user interface could be used to mark detections – both false positives and deliberate violations – as ignored, so subsequent analyses would not re-detect the same antipattern occurrences.

While this research demonstrates the efficacy of LLMs in bypassing the parsing limitations of the jOOQ DSL specifically, the underlying methodology is highly adaptable. Expanding the detection logic to encompass other query builders, such as LINQ and SQLAlchemy, would broaden the tool's applicability.

While the results of the large-scale analysis were used to find the jOOQ DSL methods most commonly associated with query antipatterns, we did not evaluate them regarding database design antipatterns. The 15 931 flagged antipattern occurrences could be further processed and analysed, seeking patterns among the detections, which could be used to improve detection heuristics in other tools, such as scripts analysing database system tables.

9 Summary

This thesis developed an automated, LLM-powered static analysis tool to detect SQL antipatterns in Java source code that uses jOOQ, and subsequently conducted a large-scale empirical analysis of their estimated prevalence.

The study began with mining 602 open-source jOOQ projects. To establish a ground truth, 61 representative projects were manually annotated, resulting in 1562 antipattern occurrences across 19 SQL antipatterns. Following statistical viability filtering, 12 antipatterns were excluded, leaving seven for all subsequent stages of the study. This filtered dataset enabled a systematic evaluation of four advanced LLMs across multiple prompting strategies. The evaluation revealed that complex reasoning prompts, such as Chain-of-Thought and Tree-of-Thought, generally failed to improve detection performance. Instead, these strategies drastically increased computational costs, extended runtimes, and in some cases degraded model stability. Consequently, the final detection tool was engineered using Claude Opus 4.5 without reasoning, with a baseline Zero-Shot Prompting strategy, achieving a high F1-score of 0.88 in a multi-label multi-occurrence localisation task.

Deploying the finalised tool across the entire 602-project corpus flagged 15 931 occurrences of seven distinct antipatterns. The empirical data demonstrated frequent co-occurrence among detected antipatterns; notably, the "Implicit Columns" and "ID Required" antipatterns were almost ubiquitous, occurring in nearly 90 % of the analysed repositories. Furthermore, the analysis revealed that the vast majority of the detected query antipattern occurrences were associated with a few specific jOOQ convenience methods, such as `selectFrom` and `Field.like`.

To ensure repeatability and facilitate future research, we open-sourced all artefacts produced as parts of the thesis. The finalised tool is available in the following GitHub repository: https://github.com/kristoisberg/jooq-antipattern-detector. All other artefacts, including

experimental scripts and their results, evaluated prompts, and source code for this document, are available in the following GitHub repository: https://github.com/kristoisberg/masters-thesis.

Ultimately, this study highlights the potential of LLMs to address the parsing limitations of dynamic, DSL-based database frameworks. By detailing the convenience methods most commonly observed alongside detected antipatterns, the research provides actionable insights to help developers avoid common database pitfalls and recognise recurring design risks in their codebases.

References

[1] *Technology | 2025 Stack Overflow Developer Survey*. 2025. URL: https://survey.stackoverflow.co/2025/technology (visited on 12/15/2025).

[2] *DB-Engines Ranking*. 2026. URL: https://db-engines.com/en/ranking (visited on 03/13/2026).

[3] Bill Karwin. *SQL antipatterns, volume 1*. Raleigh, NC: Pragmatic Programmers, Jan. 2023. ISBN: 978-1680508987.

[4] Biruk Asmare Muse et al. "On the Prevalence, Impact, and Evolution of SQL Code Smells in Data-Intensive Systems". In: *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. Ed. by Sunghun Kim et al. ACM, 2020, pp. 327–338. DOI: 10.1145/3379597.3387467.

[5] Tushar Sharma et al. "Smelly relations: measuring and understanding database schema quality". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Frances Paulisch and Jan Bosch. ACM, 2018, pp. 55–64. DOI: 10.1145/3183519.3183529.

[6] Yingjun Lyu, Ali Alotaibi, and William G. J. Halfond. "Quantifying the Performance Impact of SQL Antipatterns on Mobile Applications". In: *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 2019, pp. 53–64. DOI: 10.1109/ICSME.2019.00015.

[7] Prashanth Dintyala, Arpit Narechania, and Joy Arulraj. "SQLCheck: Automated Detection and Diagnosis of SQL Anti-Patterns". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier et al. ACM, 2020, pp. 2331–2345. DOI: 10.1145/3318464.3389754.

[8] Alan R. Hevner et al. "Design science in information systems research". In: *MIS Q.* 28.1 (Mar. 2004), pp. 75–105. ISSN: 0276-7783.

[9] *Gemini 3.1 Pro*. 2026. URL: https://deepmind.google/models/model-cards/gemini-3-1-pro/ (visited on 05/03/2026).

[10] *Google NotebookLM | AI Research Tool Thinking Partner*. 2026. URL: https://notebooklm.google/ (visited on 05/03/2026).

[11] *Lightweight coding agent that runs in your terminal*. 2026. URL: https://github.com/openai/codex (visited on 05/03/2026).

[12] *Gemini 3 Flash Model Card*. 2025. URL: https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf (visited on 05/03/2026).

[13] *Project Jupyter | Home*. 2026. URL: https://jupyter.org/ (visited on 05/10/2026).

[14] Bader Alshemaimri et al. "A survey of problematic database code fragments in software systems". In: *Engineering Reports* 3.10 (2021), e12441. DOI: 10.1002/eng2.12441.

[15] Andrew Koenig. "Patterns and Antipatterns". In: *J. Object Oriented Program.* 8.1 (1995), pp. 46–48.

[16] Scott W Ambler. *SIGS: Managing object technology: Process patterns: Building large-scale systems using object technology series number 15*. Managing object technology series. Cambridge, England: Cambridge University Press, Oct. 1998. ISBN: 0521645689.

[17] William J Brown et al. *AntiPatterns*. Nashville, TN: John Wiley & Sons, Mar. 1998. ISBN: 0471197130.

[18] Martin Fowler. *Refactoring*. 2nd ed. Addison-Wesley Signature Series (Fowler). Boston, MA: Addison Wesley, Jan. 2019. ISBN: 978-0134757599.

[19] Erki Eessaar. "On the Design of Base Tables in the SQL Databases of Some Existing Software". In: *Software Engineering Perspectives in Systems*. Ed. by Radek Silhavy. Cham: Springer International Publishing, 2022, pp. 309–324. ISBN: 978-3-031-09070-7. DOI: 10.1007/978-3-031-09070-7_26.

[20] Erki Eessaar. "On the Naming of Database Objects in the SQL Databases of Some Existing Software". In: *Software Engineering Research in System Science*. Ed. by Radek Silhavy and Petr Silhavy. Cham: Springer International Publishing, 2023, pp. 534–550. ISBN: 978-3-031-35311-6. DOI: 10.1007/978-3-031-35311-6_51.

[21] Phil Factor. "SQL Code Smells". In: *Red Gate Software Ltd* (2014).

[22] Jimmy Angelakos. *PostgreSQL mistakes and how to avoid them*. New York, NY: Manning Publications, July 2025. ISBN: 978-1633436879.

[23] Bill Karwin. *More SQL Antipatterns*. Raleigh, NC: Pragmatic Programmers, June 2026. ISBN: 979-8888652060.

[24] *More SQL Antipatterns: Avoid Common But Deadly Mistakes of Database Optimization - Bill Karwin - Google Books*. 2026. URL: https://books.google.ee/books/about/More_SQL_Antipatterns.html?id=d2H00QEACAAJ (visited on 05/06/2026).

[25] Erki Eessaar. *Teema 10. Loogiline disain. CASE vahendid*. Tallinn University of Technology, Andmebaasid I Lecture Notes. 2026. URL: https://maurus.ttu.ee/download.php?aine=346&document=32235&tyyp=do (visited on 03/15/2026).

[26] *jOOQ: The easiest way to write SQL in Java*. 2026. URL: https://www.jooq.org/ (visited on 05/02/2026).

[27] *Your relational data. Objectively. - Hibernate ORM*. 2026. URL: https://hibernate.org/orm/ (visited on 05/02/2026).

[28] *SQL building*. 2025. URL: https://www.jooq.org/doc/latest/manual/sql-building/ (visited on 10/07/2025).

[29] *Code generation*. 2025. URL: https://www.jooq.org/doc/latest/manual/code-generation/ (visited on 10/07/2025).

[30] *SQL execution*. 2025. URL: https://www.jooq.org/doc/latest/manual/sql-execution/ (visited on 10/11/2025).

[31] *jOOQ*. 2025. URL: https://github.com/jOOQ/jOOQ (visited on 10/11/2025).

[32] *Some of our customers*. 2025. URL: https://www.jooq.org/customers (visited on 10/11/2025).

[33] Vlad Mihalcea. *High-Performance Java Persistence*. Vlad Mihalcea, Oct. 2016. ISBN: 978-9730228236.

[34] *SonarQube: Fight AI Slop Verify AI Code | Sonar*. 2026. URL: https://www.sonarsource.com/products/sonarqube/ (visited on 05/02/2026).

[35] *PMD*. 2026. URL: https://pmd.github.io/ (visited on 05/02/2026).

[36] Radu Marinescu. "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws". In: *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*. IEEE Computer Society, 2004, pp. 350–359. DOI: 10.1109/ICSM.2004.1357820.

[37] Arthur J Riel. *Object-oriented design heuristics*. Boston, MA: Addison-Wesley Educational, Apr. 1996. ISBN: 978-0321774965.

[38] Naouel Moha et al. "DECOR: A Method for the Specification and Detection of Code and Design Smells". In: *IEEE Trans. Software Eng.* 36.1 (2010), pp. 20–36. DOI: 10.1109/TSE.2009.50.

[39] Thanis Paiva et al. "On the evaluation of code smells and detection tools". In: *J. Softw. Eng. Res. Dev.* 5 (2017), p. 7. DOI: 10.1186/S40411-017-0041-1.

[40] Brittany Johnson et al. "Why don't software developers use static analysis tools to find bugs?" In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. Ed. by David Notkin, Betty H. C. Cheng, and Klaus Pohl. IEEE Computer Society, 2013, pp. 672–681. DOI: 10.1109/ICSE.2013.6606613.

[41] Francesca Arcelli Fontana et al. "Comparing and experimenting machine learning techniques for code smell detection". In: *Empir. Softw. Eng.* 21.3 (2016), pp. 1143–1191. DOI: 10.1007/S10664-015-9378-4.

[42] Ruchika Malhotra, Bhawna Jain, and Marouane Kessentini. "Examining deep learning's capability to spot code smells: a systematic literature review". In: *Clust. Comput.* 26.6 (2023), pp. 3473–3501. DOI: 10.1007/S10586-023-04144-1.

[43] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: *CoRR* abs/2107.03374 (2021). arXiv: 2107.03374.

[44] Tianyang Liu, Canwen Xu, and Julian J. McAuley. “RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems”. In: *CoRR* abs/2306.03091 (2023). DOI: 10.48550/ARXIV.2306.03091. arXiv: 2306.03091.

[45] *AI Code Reviews | CodeRabbit | Try for Free*. 2026. URL: https://www.coderabbit.ai/ (visited on 05/02/2026).

[46] Robert Blaauwendraad. “Evaluating Large Language Models for Code Smell Detection”. Bachelor’s Thesis. Radboud University, 2025. URL: https://www.cs.ru.nl/bachelors-theses/2025/Robert_Blaauwendraad___1084960___Evaluating_Large_Language_Models_for_Code_Smell_Detection.pdf (visited on 12/16/2025).

[47] Ahmed R. Sadik and Siddhata Govind. “Benchmarking LLM for Code Smells Detection: OpenAI GPT-4.0 vs DeepSeek-V3”. In: *CoRR* abs/2504.16027 (2025). DOI: 10.48550/ARXIV.2504.16027. arXiv: 2504.16027.

[48] Vinicius Sousa et al. “Effectiveness of Small and Large Language Models for PL/SQL Bad Smell Detection”. In: *Anais do XL Simpósio Brasileiro de Bancos de Dados*. Fortaleza/CE: SBC, 2025, pp. 399–412. DOI: 10.5753/sbbd.2025.247256.

[49] Pengfei Liu et al. “Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing”. In: *ACM Comput. Surv.* 55.9 (2023), 195:1–195:35. DOI: 10.1145/3560815.

[50] Jules White et al. “A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT”. In: *CoRR* abs/2302.11382 (2023). DOI: 10.48550/ARXIV.2302.11382. arXiv: 2302.11382.

[51] Pranab Sahoo et al. “A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications”. In: *CoRR* abs/2402.07927 (2024). DOI: 10.48550/ARXIV.2402.07927. arXiv: 2402.07927.

[52] Takeshi Kojima et al. “Large Language Models are Zero-Shot Reasoners”. In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. Ed. by Sanmi Koyejo et al. 2022.

[53] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle et al. 2020.

[54] Jason Wei et al. “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”. In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. Ed. by Sanmi Koyejo et al. 2022.

[55] Lennart Meincke et al. “Prompting Science Report 2: The Decreasing Value of Chain of Thought in Prompting”. In: *CoRR* abs/2506.07142 (2025). DOI: 10.48550/ARXIV.2506.07142. arXiv: 2506.07142.

[56] Dave Hulbert. *Using Tree-of-Thought Prompting to boost ChatGPT's reasoning*. https://github.com/dave1010/tree-of-thought-prompting. May 2023. DOI: 10.5281/ZENODO.10323452.

[57] Jieyi Long. "Large Language Model Guided Tree-of-Thought". In: *CoRR* abs/2305.08291 (2023). DOI: 10.48550/ARXIV.2305.08291. arXiv: 2305.08291.

[58] Shunyu Yao et al. "Tree of Thoughts: Deliberate Problem Solving with Large Language Models". In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. Ed. by Alice Oh et al. 2023.

[59] *DbDeo - Database smell detector*. 2018. URL: https://github.com/tushartushar/DbDeo (visited on 05/02/2026).

[60] Csaba Nagy and Anthony Cleve. "A Static Code Smell Detector for SQL Queries Embedded in Java Code". In: *17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017, Shanghai, China, September 17-18, 2017*. IEEE Computer Society, 2017, pp. 147–152. DOI: 10.1109/SCAM.2017.19.

[61] Csaba Nagy and Anthony Cleve. "SQLInspect: a static analyzer to inspect database usage in Java applications". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Michel Chaudron et al. ACM, 2018, pp. 93–96. DOI: 10.1145/3183440.3183496.

[62] *jOOQ as a SQL builder with code generation*. 2025. URL: https://www.jooq.org/doc/latest/manual/getting-started/use-cases/jooq-as-a-sql-builder-with-code-generation/ (visited on 10/11/2025).

[63] Francisco Gonçalves de Almeida Filho et al. "Prevalence of bad smells in PL/SQL projects". In: *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*. Ed. by Yann-Gaël Guéhéneuc, Foutse Khomh, and Federica Sarro. IEEE / ACM, 2019, pp. 116–121. DOI: 10.1109/ICPC.2019.00025.

[64] *Automatically identify anti-patterns in SQL queries*. 2024. URL: https://github.com/jarulraj/sqlcheck (visited on 05/02/2026).

[65] Erki Eessaar. "Automating Detection of Occurrences of PostgreSQL Database Design Problems". In: *Databases and Information Systems*. Ed. by Tarmo Robal et al. Cham: Springer International Publishing, 2020, pp. 176–189. ISBN: 978-3-030-57672-1.

[66] Erki Eessaar. "The Usage of Declarative Integrity Constraints in the SQL Databases of Some Existing Software". In: *Software Engineering and Algorithms*. Ed. by Radek Silhavy. Cham: Springer International Publishing, 2021, pp. 375–390. ISBN: 978-3-030-77442-4.

[67] Poonyanuch Khumnin and Twittie Senivongse. “SQL antipatterns detection and database refactoring process”. In: *18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2017, Kanazawa, Japan, June 26-28, 2017*. Ed. by Teruhisa Hochin, Hiroaki Hirata, and Hiroki Nomiya. IEEE Computer Society, 2017, pp. 199–205. DOI: 10.1109/SNPD.2017.8022723.

[68] *Code generation and version control*. 2025. URL: https://www.jooq.org/doc/latest/manual/code-generation/codegen-version-control/ (visited on 10/02/2025).

[69] *REST API endpoints for search*. 2022. URL: https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28 (visited on 12/15/2025).

[70] *Welcome to Apache Maven – Maven*. 2026. URL: https://maven.apache.org/ (visited on 05/02/2026).

[71] *Gradle Build Tool*. 2026. URL: https://gradle.org/ (visited on 05/02/2026).

[72] *Gemini 3 Pro Model Card*. 2025. URL: https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf (visited on 05/03/2026).

[73] *Baeldung*. 2026. URL: https://www.baeldung.com/ (visited on 05/09/2026).

[74] Miltiadis Allamanis and Charles Sutton. “Mining source code repositories at massive scale using language modeling”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR ’13, San Francisco, CA, USA, May 18-19, 2013*. Ed. by Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim. IEEE Computer Society, 2013, pp. 207–216. DOI: 10.1109/MSR.2013.6624029.

[75] Mathieu Goeminne and Tom Mens. “Towards a survival analysis of database framework usage in Java projects”. In: *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*. Ed. by Rainer Koschke, Jens Krinke, and Martin P. Robillard. IEEE Computer Society, 2015, pp. 551–555. DOI: 10.1109/ICSM.2015.7332512.

[76] *GH Archive*. 2025. URL: https://www.gharchive.org/ (visited on 05/02/2026).

[77] *Running this query is gonna cost me* 180. 2025. URL: https://www.linkedin.com/posts/yamiteru_running-this-query-is-gonna-cost-me-180-activity-7400177031298048001-joHQ/ (visited on 03/07/2026).

[78] Nuthan Munaiah et al. “Curating GitHub for engineered software projects”. In: *Empir. Softw. Eng.* 22.6 (2017), pp. 3219–3253. DOI: 10.1007/S10664-017-9512-6.

[79] Georgios Gousios. “The GHTorent dataset and tool suite”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR ’13, San Francisco, CA, USA, May 18-19, 2013*. Ed. by Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim. IEEE Computer Society, 2013, pp. 233–236. DOI: 10.1109/MSR.2013.6624034.

[80] *GHTorrent used to be largest github archive. It was stopped updating couple of years ago and now whole site is down. Any alternatives?* 2024. URL: https://www.reddit.com/r/DataHoarder/comments/1d1tuek/ghtorrent_used_to_be_largest_github_archive_it/ (visited on 03/07/2026).

[81] *SQL: SELECT **. 2025. URL: https://www.jooq.org/doc/latest/manual/reference/dont-do-this/dont-do-this-sql-select-all/ (visited on 09/29/2025).

[82] Markus Winand. *SQL Performance Explained*. Vienna, Austria: Winand, Markus, July 2012. ISBN: 978-3950307825.

[83] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. "NullAway: practical type-based null safety for Java". In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by Marlon Dumas et al. ACM, 2019, pp. 740–750. DOI: 10.1145/3338906.3338919.

[84] Henrietta Dombrovskaya, Boris Novikov, and Anna Bailliekova. "Long Queries: Additional Techniques". In: *PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries*. Berkeley, CA: Apress, 2024, pp. 147–174. ISBN: 979-8-8688-0069-6. DOI: 10.1007/979-8-8688-0069-6_7.

[85] *ExceptionHandler (Spring Framework 7.0.7 API)*. 2026. URL: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ExceptionHandler.html (visited on 05/03/2026).

[86] *Error (micronaut-parent 4.10.16 API)*. 2026. URL: https://docs.micronaut.io/4.10.16/api/io/micronaut/http/annotation/Error.html (visited on 05/03/2026).

[87] J. Richard Landis and Gary G. Koch. "The Measurement of Observer Agreement for Categorical Data". In: *Biometrics* 33.1 (1977).

[88] *AI Model & API Providers Analysis*. 2025. URL: https://artificialanalysis.ai/ (visited on 12/16/2025).

[89] *Introducing Structured Outputs in the API*. Aug. 2024. URL: https://openai.com/index/introducing-structured-outputs-in-the-api/ (visited on 03/06/2026).

[90] *Update to GPT-5 System Card: GPT-5.2*. 2025. URL: https://deploymentsafety.openai.com/gpt-5-2/introduction (visited on 05/03/2026).

[91] GLM. "GLM-5: from Vibe Coding to Agentic Engineering". In: *CoRR* abs/2602.15763 (2026). DOI: 10.48550/ARXIV.2602.15763. arXiv: 2602.15763.

[92] *System Card: Claude Opus 4.5*. 2025. URL: https://www-cdn.anthropic.com/bf10f64990cfda0ba858290be7b8cc6317685f47.pdf (visited on 05/03/2026).

[93] OpenAI. "gpt-oss-120b & gpt-oss-20b Model Card". In: *CoRR* abs/2508.10925 (2025). DOI: 10.48550/ARXIV.2508.10925. arXiv: 2508.10925.

[94] GLM. "GLM-4.5: Agentic, Reasoning, and Coding (ARC) Foundation Models". In: *CoRR* abs/2508.06471 (2025). DOI: 10.48550/ARXIV.2508.06471. arXiv: 2508.06471.

[95] *System Card: Claude Opus 4.6*. 2026. URL: https://www-cdn.anthropic.com/0dd865075ad3132672ee0ab40b05a53f14cf5288.pdf (visited on 05/03/2026).

[96] Zizheng Zhan et al. "KAT-Coder Technical Report". In: *CoRR* abs/2510.18779 (2025). DOI: 10.48550/ARXIV.2510.18779. arXiv: 2510.18779.

[97] Kimi Team. "Kimi K2: Open Agentic Intelligence". In: *CoRR* abs/2507.20534 (2025). DOI: 10.48550/ARXIV.2507.20534. arXiv: 2507.20534.

[98] *The official Python library for the OpenAI API*. 2026. URL: https://github.com/openai/openai-python (visited on 05/03/2026).

[99] *OpenRouter*. 2026. URL: https://openrouter.ai/ (visited on 05/03/2026).

[100] *GPT-5.4 Thinking System Card*. 2026. URL: https://deploymentsafety.openai.com/gpt-5-4-thinking (visited on 05/03/2026).

[101] *Bun – A fast all-in-one JavaScript runtime*. 2026. URL: https://bun.com/ (visited on 05/03/2026).

[102] *TypeScript: JavaScript with syntax for types.* 2026. URL: https://www.typescriptlang.org/ (visited on 05/03/2026).

[103] *Biome, toolchain of the web*. 2026. URL: https://biomejs.dev/ (visited on 05/03/2026).

[104] *AI SDK*. 2026. URL: https://ai-sdk.dev/ (visited on 05/03/2026).

[105] *node.js command-line interfaces made easy*. 2026. URL: https://github.com/tj/commander.js/ (visited on 05/03/2026).

[106] *It's a very fast and efficient glob library for Node.js*. 2026. URL: https://github.com/mrmlnc/fast-glob (visited on 05/03/2026).

[107] *Run multiple promise-returning async functions with limited concurrency*. 2026. URL: https://github.com/sindresorhus/p-limit (visited on 05/03/2026).

[108] *Zod*. 2026. URL: https://zod.dev/ (visited on 05/03/2026).

[109] Kelly Hong, Anton Troynikov, and Jeff Huber. *Context Rot: How Increasing Input Tokens Impacts LLM Performance*. Tech. rep. Chroma, July 2025. URL: https://trychroma.com/research/context-rot.

[110] Loup Meurice, Csaba Nagy, and Anthony Cleve. "Static Analysis of Dynamic Database Usage in Java Systems". In: *Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings*. Ed. by Selmin Nurcan et al. Vol. 9694. Lecture Notes in Computer Science. Springer, 2016, pp. 491–506. DOI: 10.1007/978-3-319-39696-5_30.

[111] Karl Tamberg and Hayretdin Bahsi. "Suurte keelemudelite kasutamine turvanõrkuste tuvastamiseks lähtekoodis: põhjalik võrdlusuuring". Master's Thesis. May 2024. URL: https://digikogu.taltech.ee/et/item/a0cfeda0-dc43-4408-8116-2fb657ed9a44 (visited on 03/02/2026).

[112] Tarmo Ploom and Erki Eessaar. “Suurte keelemudelite rakendamine mudelite halbade lõhnade tuvastamiseks UML skeemides”. Bachelor’s Thesis. Jan. 2026. URL: https://digikogu.taltech.ee/et/item/dbbda967-3e60-414f-bc99-d33d279c4eb8 (visited on 03/18/2026).

[113] David A Kolb. *Experiential learning*. Upper Saddle River, NJ: Financial Times Prentice Hall, Oct. 1983. ISBN: 978-0132952613.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Kristo Isberg

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Detecting SQL Antipatterns in jOOQ Database Access Code Using Large Language Models", supervised by Erki Eessaar
    1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
    1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright
2. I am aware that the author also retains the rights specified in clause 1 of the nonexclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

11.05.2026

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive licence shall not be valid for the period.

Appendix 2 – Search terms used to find projects from GitHub

```
# Maven
language:xml "<artifactId>jooq</artifactId>"
language:xml "<artifactId>jooq-"
language:xml "<artifactId>jooq-codegen-maven</artifactId>"
language:xml "<artifactId>jooq-codegen</artifactId>"
language:xml "<artifactId>jooq-meta/artifactId>"
language:xml "<artifactId>jooq-meta-extensions</artifactId>"
language:xml "<artifactId>jooq-meta-extensions-liquibase</artifactId>"
language:xml "<artifactId>jooq-meta-extensions-hibernate</artifactId>"
language:xml "<artifactId>jooq-postgres-extensions</artifactId>"
language:xml "<artifactId>jooq-jackson-extensions</artifactId>"
language:xml "<artifactId>jooq-jpa-extensions</artifactId>"
language:xml "<artifactId>jooq-reactor-extensions</artifactId>"
language:xml "<artifactId>jooq-beans-extensions</artifactId>"
language:xml "<artifactId>jooq-checker</artifactId>"

language:"Maven POM" "<artifactId>jooq</artifactId>"
language:"Maven POM" "<artifactId>jooq-"
language:"Maven POM" "<artifactId>jooq-codegen-maven</artifactId>"
language:"Maven POM" "<artifactId>jooq-codegen</artifactId>"
language:"Maven POM" "<artifactId>jooq-meta</artifactId>"
language:"Maven POM" "<artifactId>jooq-meta-extensions</artifactId>"
language:"Maven POM" "<artifactId>jooq-meta-extensions-liquibase</
artifactId>"
language:"Maven POM" "<artifactId>jooq-meta-extensions-hibernate</
artifactId>"
language:"Maven POM" "<artifactId>jooq-postgres-extensions</artifactId
>"
language:"Maven POM" "<artifactId>jooq-jackson-extensions</artifactId>"
language:"Maven POM" "<artifactId>jooq-jpa-extensions</artifactId>"
language:"Maven POM" "<artifactId>jooq-reactor-extensions</artifactId>"
language:"Maven POM" "<artifactId>jooq-beans-extensions</artifactId>"
language:"Maven POM" "<artifactId>jooq-checker</artifactId>"

# Gradle short syntax
language:gradle "\"org.jooq:jooq:"
language:gradle "'org.jooq:jooq:"
language:gradle "\"org.jooq:jooq-"
language:gradle "'org.jooq:jooq-"
language:gradle "\"org.jooq:jooq-meta"
```

```
language:gradle "'org.jooq:jooq-meta"
language:gradle "\"org.jooq:jooq"
language:gradle "'org.jooq:jooq"

# Gradle long syntax
language:gradle "group: \"org.jooq\""
language:gradle "group:\"org.jooq\""
language:gradle "group: 'org.jooq'"
language:gradle "group:'org.jooq'"

# Gradle plugins
language:gradle "jooq-codegen-gradle"
language:gradle "\"nu.studer.jooq\""
language:gradle "'nu.studer.jooq'"

# Gradle Kotlin syntax
extension:kts "org.jooq:jooq:"
extension:kts "org.jooq:jooq-"
extension:kts "org.jooq:jooq"
extension:kts "jooq-codegen-gradle"
extension:kts "nu.studer.jooq"
```

Figure 14. Search terms used to find projects from GitHub.

Appendix 3 – Projects omitted from the dataset

- **jOOQ/jOOQ**, **zhoupan/jooq-android**, **rubenvidalcss/test-gamma-jooq**, **cf-testorg/jOOQ-test**: the source code of jOOQ, rather than projects using jOOQ.
- **aizedevops/spring-boot-projects**, **alephzed/baeldung-tutorials**, **bkama-raj5054/tutorials1**, **chaimu100/java-test-for-codeql**, **ennva/java-tutorials-master**, **eugenp/tutorials**, **ganigapetaravali/jmeter**, **iotaplatforms/tutorials**, **iSharkFly-Docs/java-tutorials**, **joniyed/Super-Java**, **KevinOCM99/tutorials**, **kraczo/craw**, **nice1111hh/Java2**, **nicolaj0/smino**, **pivotal-mark-higuera/tutorials**, **ramesh1972/samples-biz**, **ranjan-projects/tutorials**, **Rohan-flutterint/tutorials**, **shydesky/tutorials**, **syliGaye/tutorials**, **TheShubham-K/getting-started-with-springboot**, **zubairmujeeb/spring-boot-sample-projects**, **tuyucheng7/taketoday-tutorial4j-backup**, **tuyucheng777/taketoday-tutorial4j**: repository clones of a Spring tutorial by Baeldung.
- **neekon2508/SpringStudy**, **qkrtkdwns3410/pro-spring-6**, **sivaprasadreddy/beginning-spring-boot-2**, **steveyangpi/pro-spring-6-Learning**: repository clones of a Spring tutorial by Apress.
- **devYSK/study_repo**, **hnstsn/jooq-practice**, **kyhyeok/jOOQ**, **pnci1029/mypro**, **today8934/jOOQ-first-look**: repository clones of a jOOQ tutorial by Inflearn.
- **silentbalanceyh/vertx-zero-0.6**, **51568488/vertx-zero**, **silentbalanceyh/vertx-zero**: repository clones of zero-ws/zero-ecotope.
- **jhavierc/EjemplosJava**, **optimus-alfaomega/Java-Casos-Practicos**: repository clones of picodotdev/blog-ejemplos.
- **Davincii254/Stroom**: repository clone of gchq/stroom (discovered later).

Appendix 4 – List of annotated antipatterns

Table 21. List of annotated antipatterns.

Category	**Antipattern**
Logical Database Design Antipatterns	Jaywalking
	Naive Trees
	ID Required
	Keyless Entry
	Entity-Attribute-Value
	Polymorphic Associations
	Multicolumn Attributes
	Metadata Tribbles
Physical Database Design Antipatterns	Rounding Errors
	31 Flavors
	Phantom Files
Query Antipatterns	Fear of the Unknown
	Ambiguous Groups
	Random Selection
	Poor Man's Search Engine
	Implicit Columns
Application Development Antipatterns	Readable Passwords
	SQL Injection
	Standard Operating Procedures

Appendix 5 – Decision diagrams for the antipattern annotation process

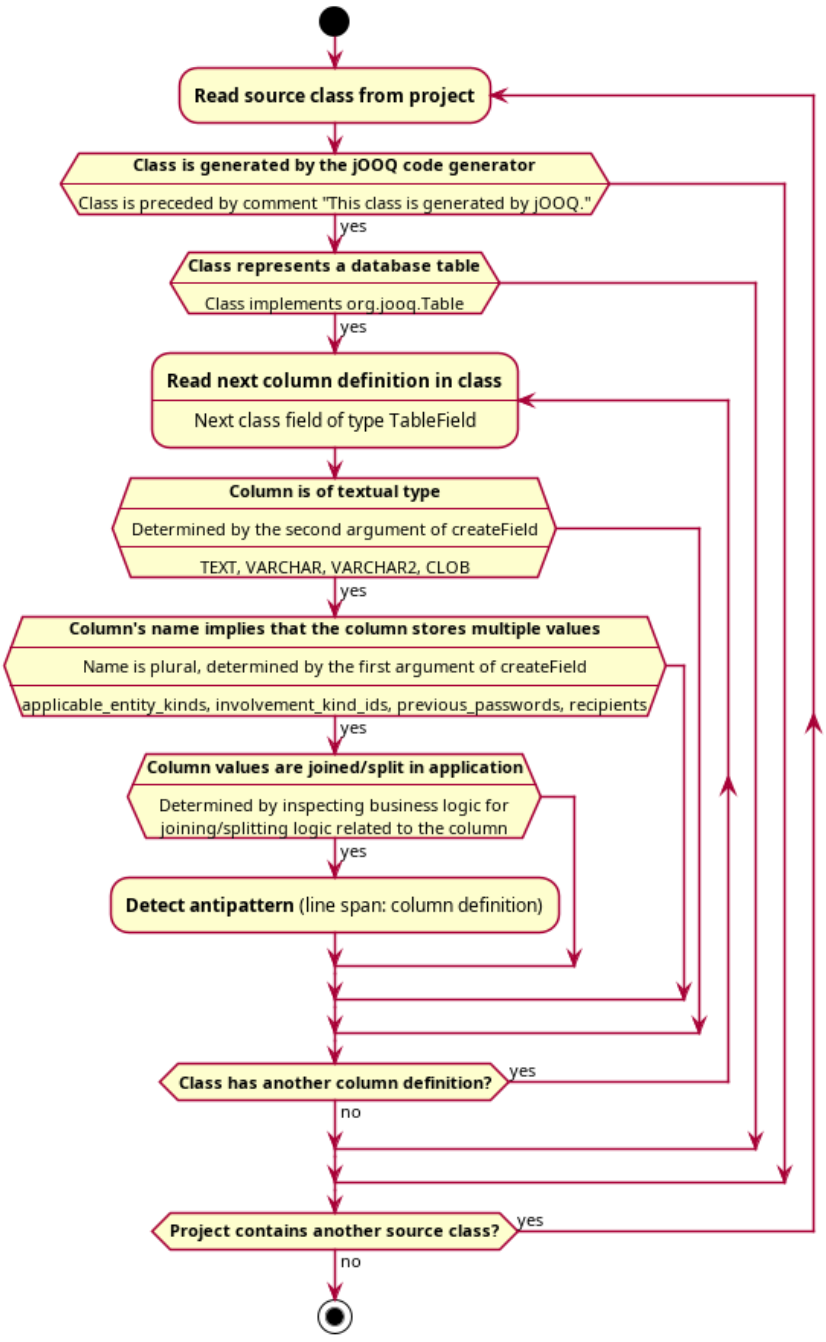

Figure 15. Decision tree for annotating the “Jaywalking” SQL antipattern.

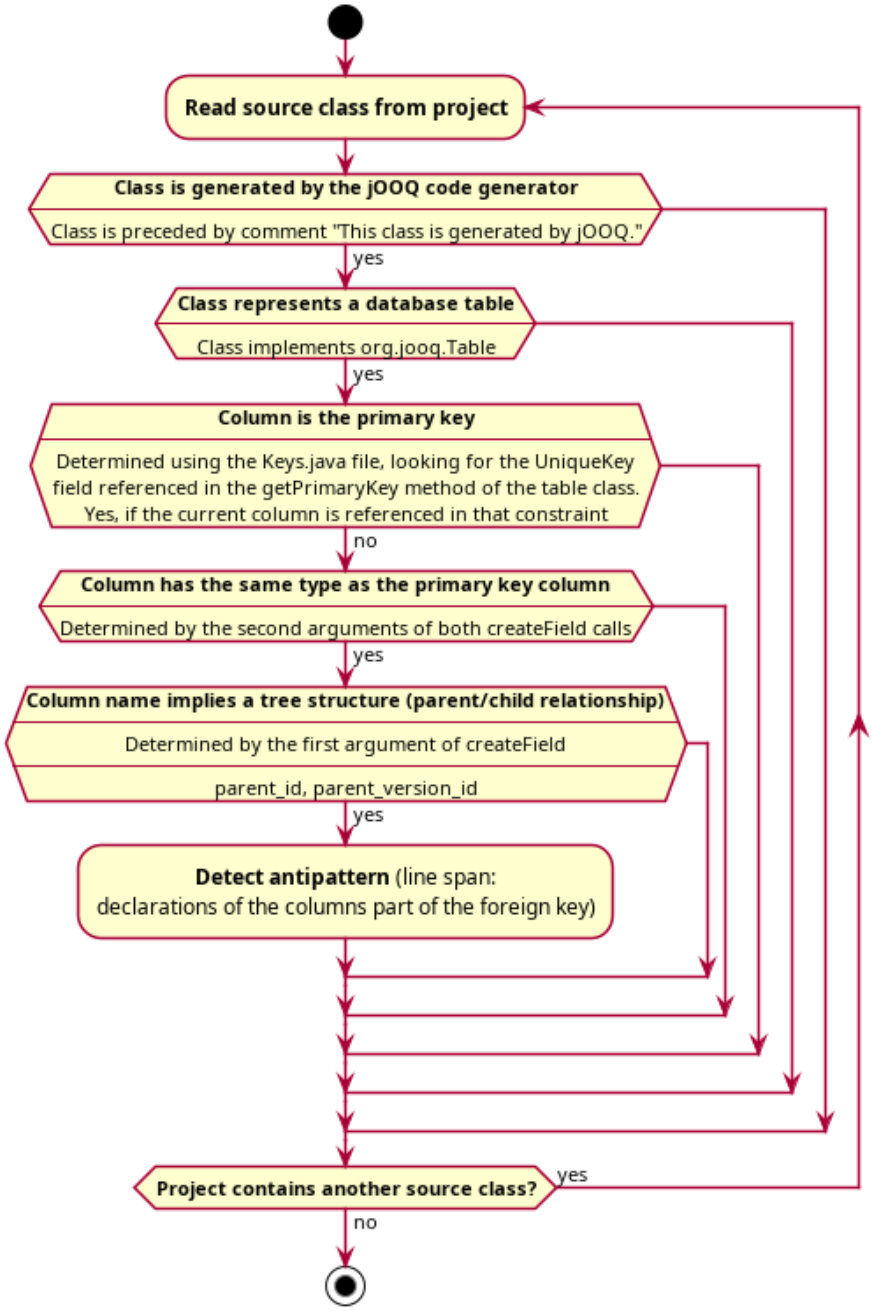

Figure 16. Decision tree for annotating the “Naive Trees” SQL antipattern.

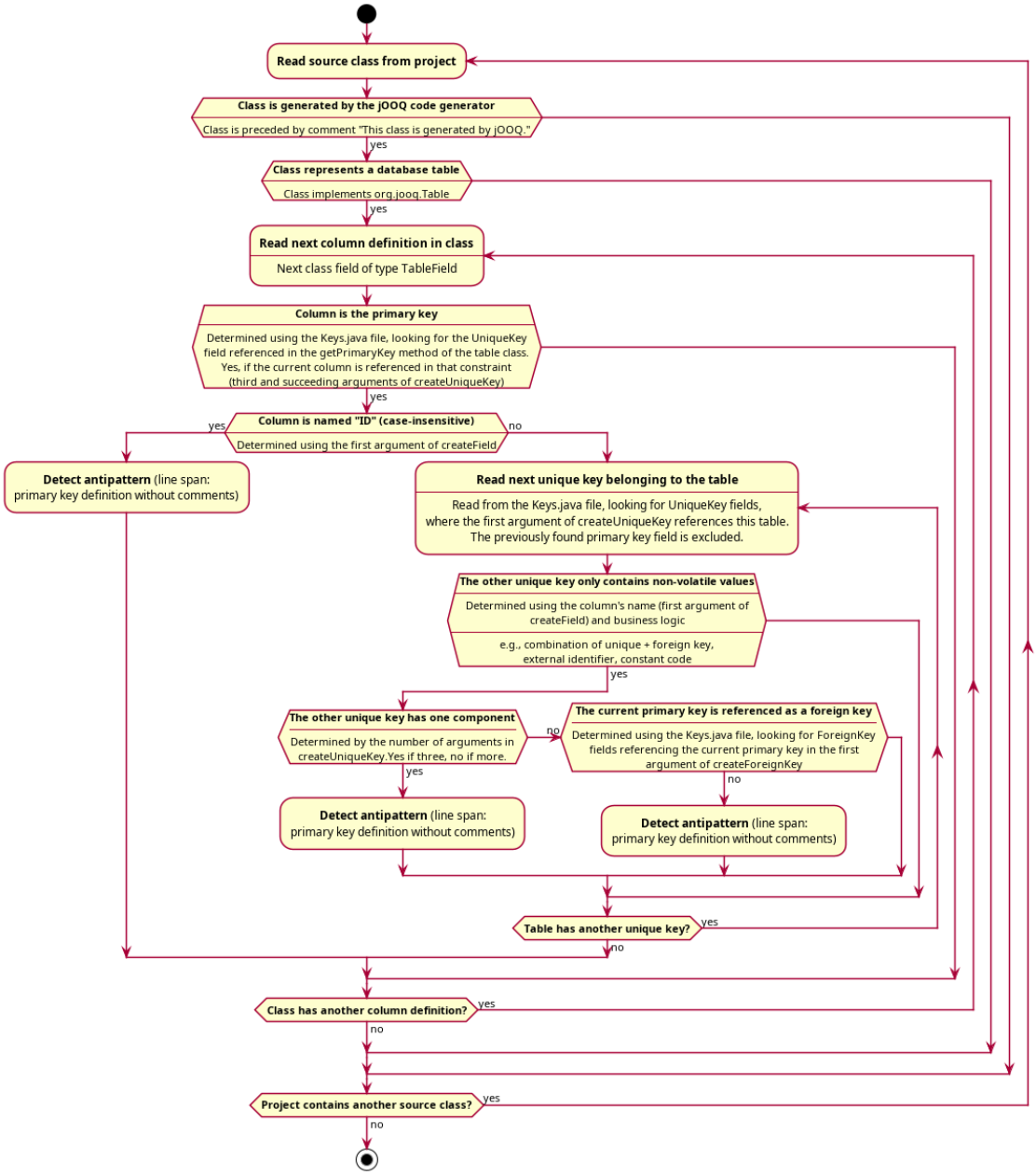

Figure 17. Decision tree for annotating the “ID Required” SQL antipattern.

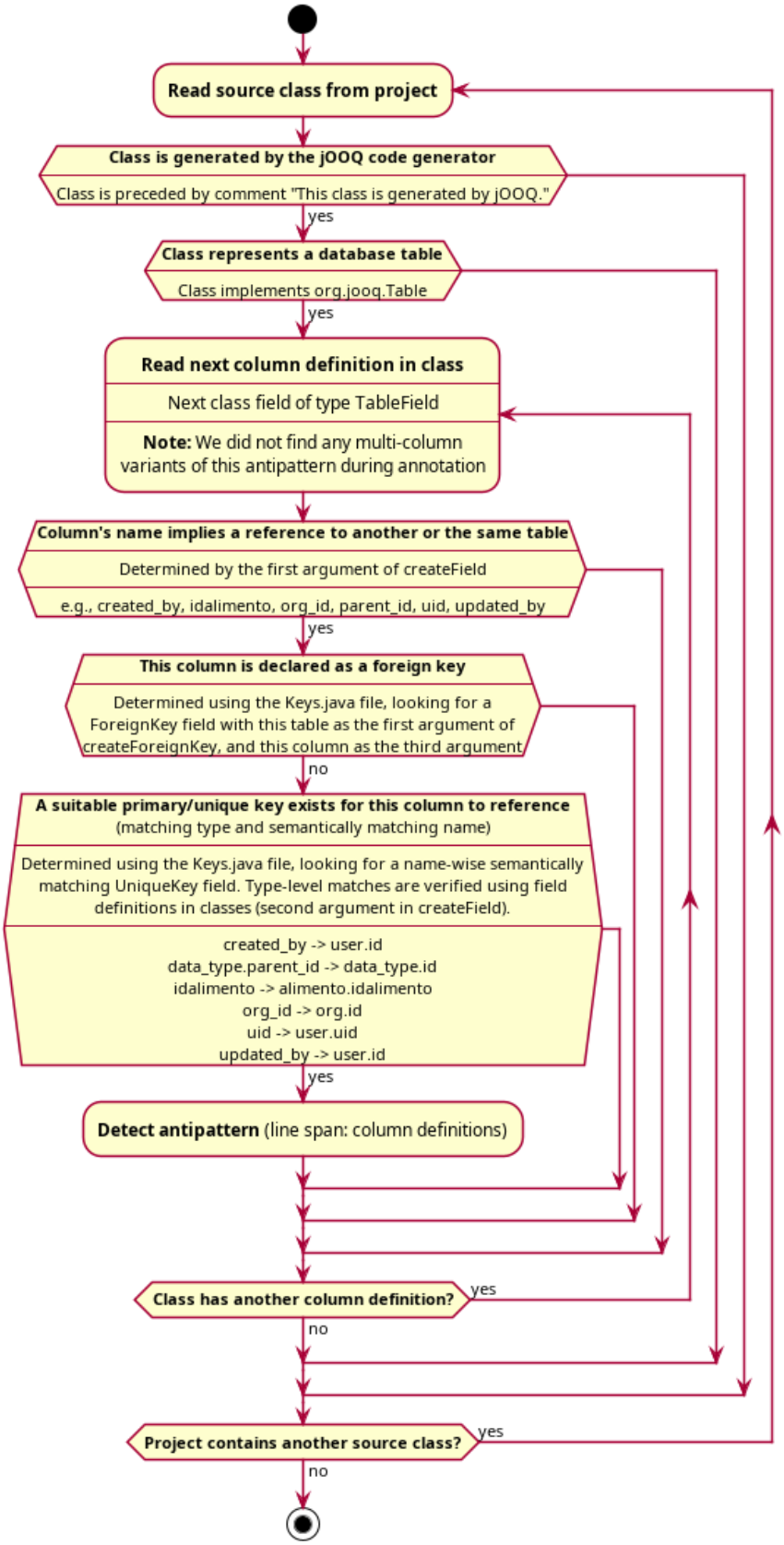

Figure 18. Decision tree for annotating the “Keyless Entry” SQL antipattern.

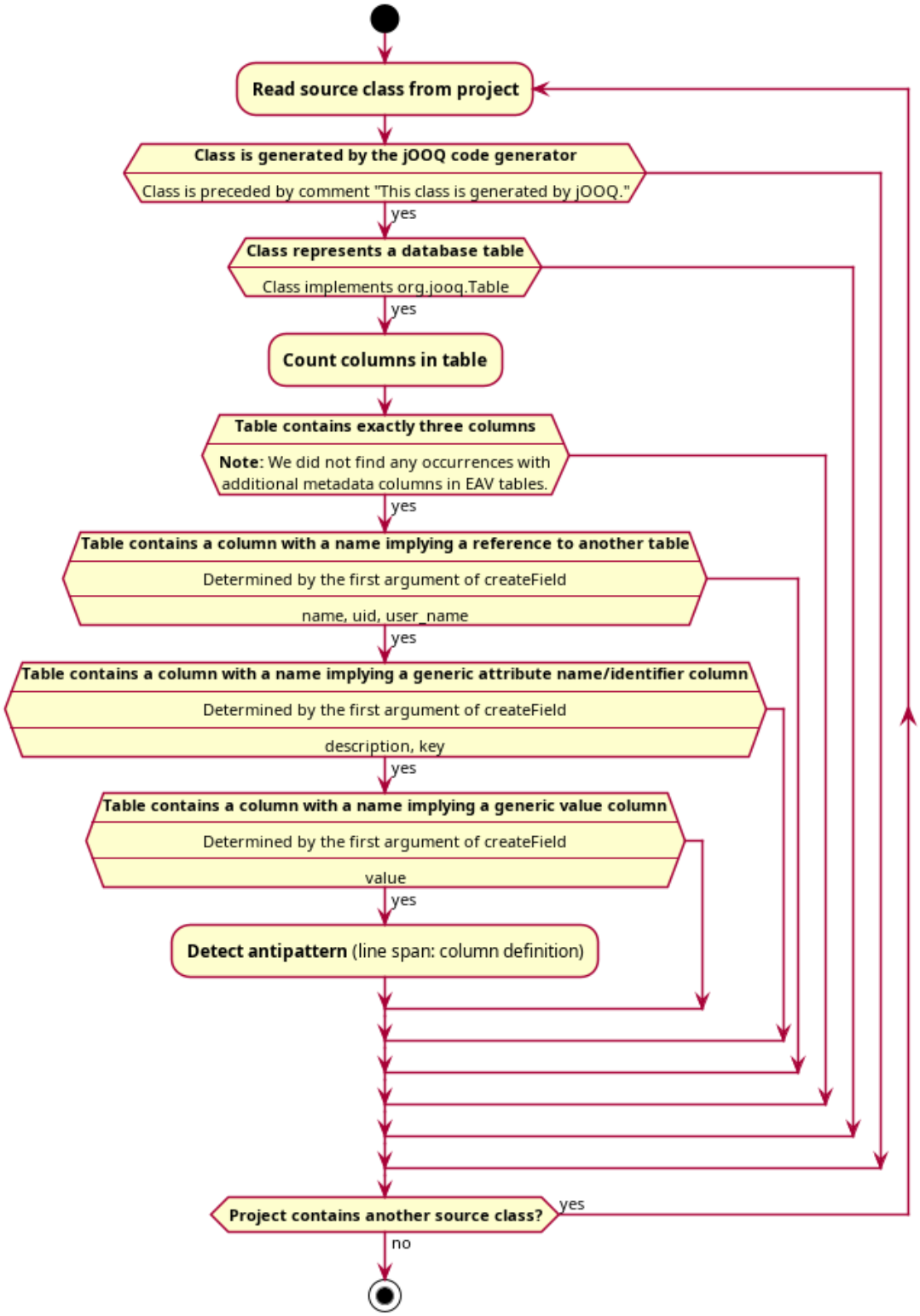

Figure 19. Decision tree for annotating the “Entity-Attribute-Value” SQL antipattern.

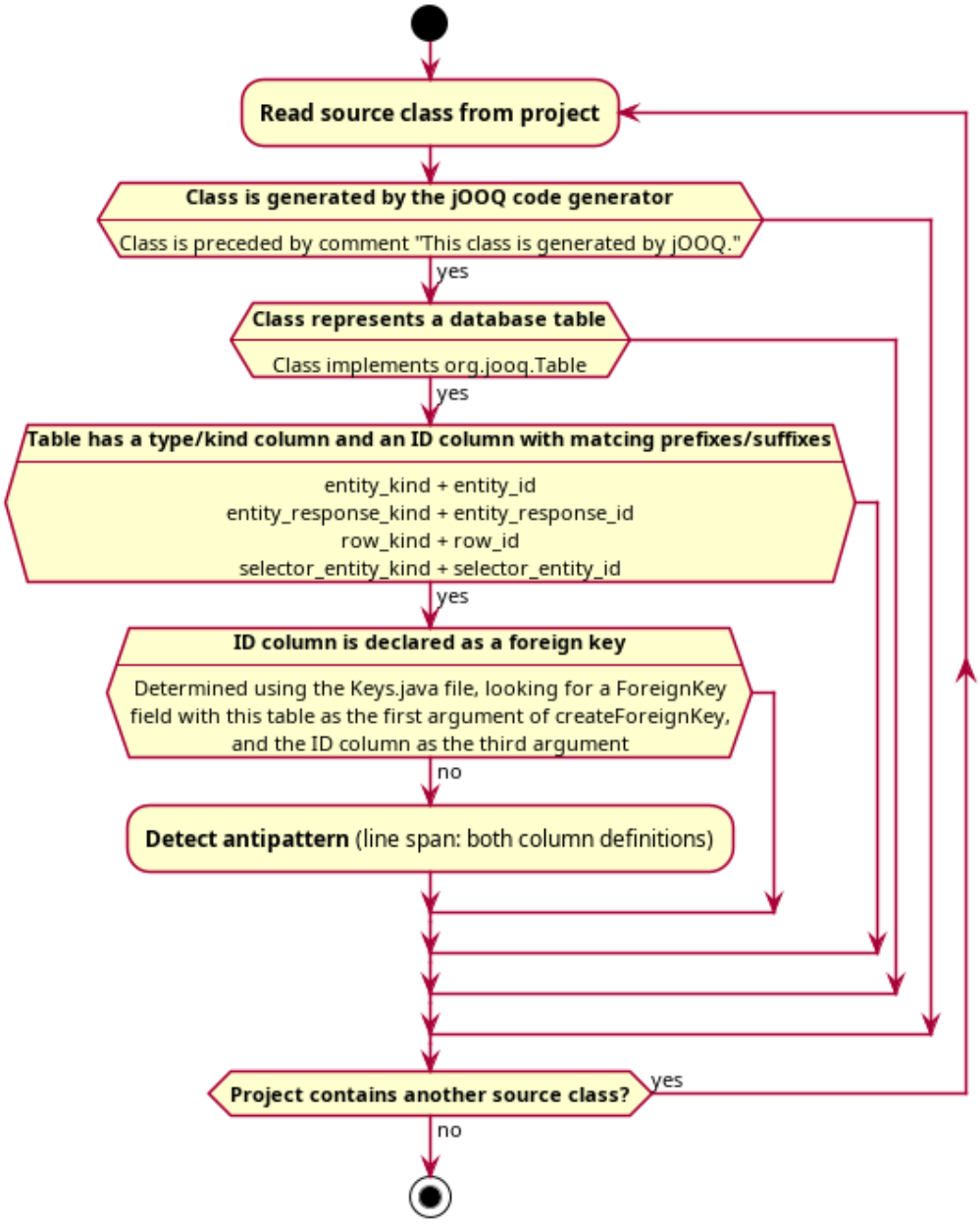

Figure 20. Decision tree for annotating the “Polymorphic Associations” SQL antipattern.

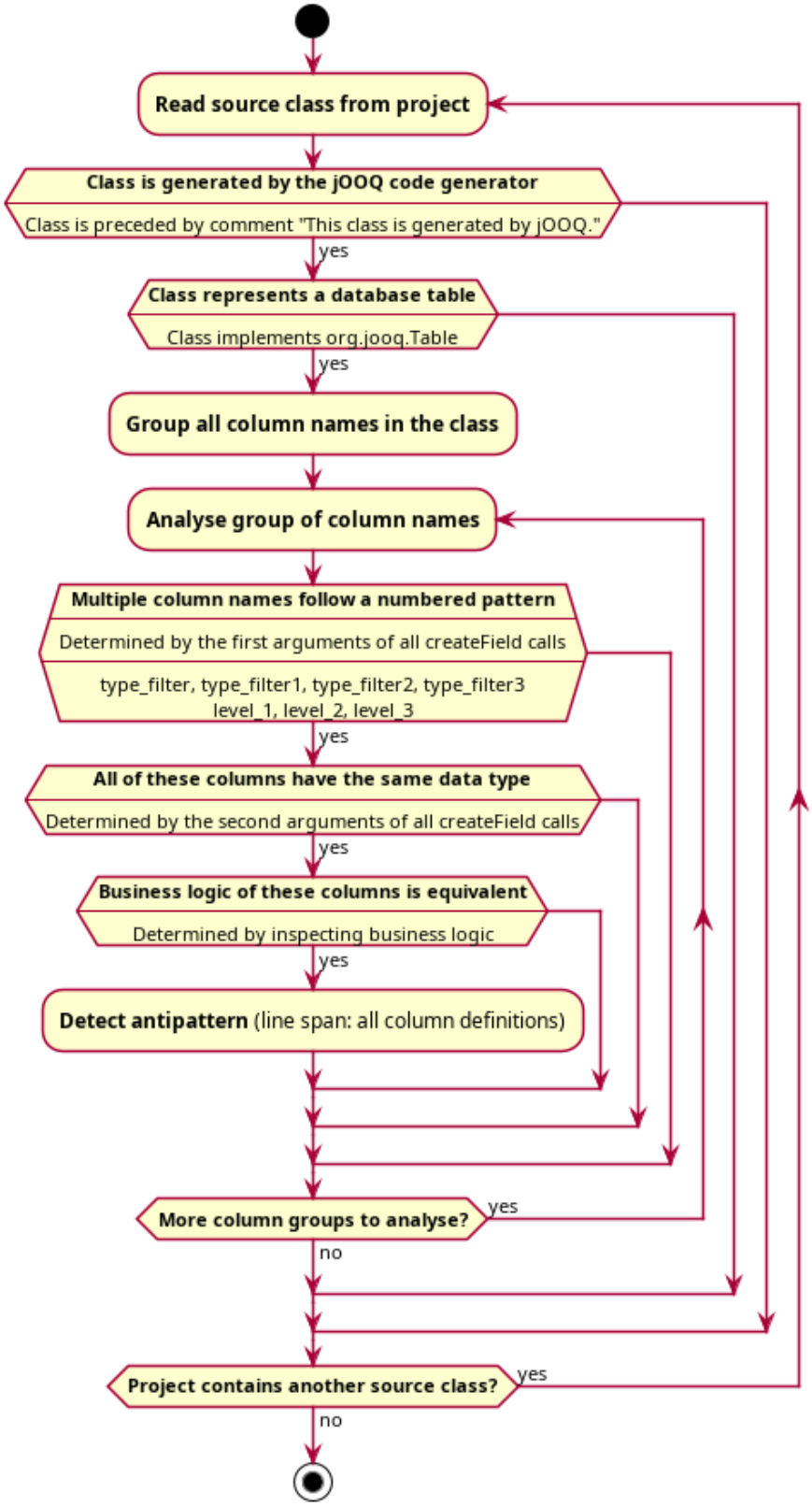

Figure 21. Decision tree for annotating the “Multicolumn Attributes” SQL antipattern.

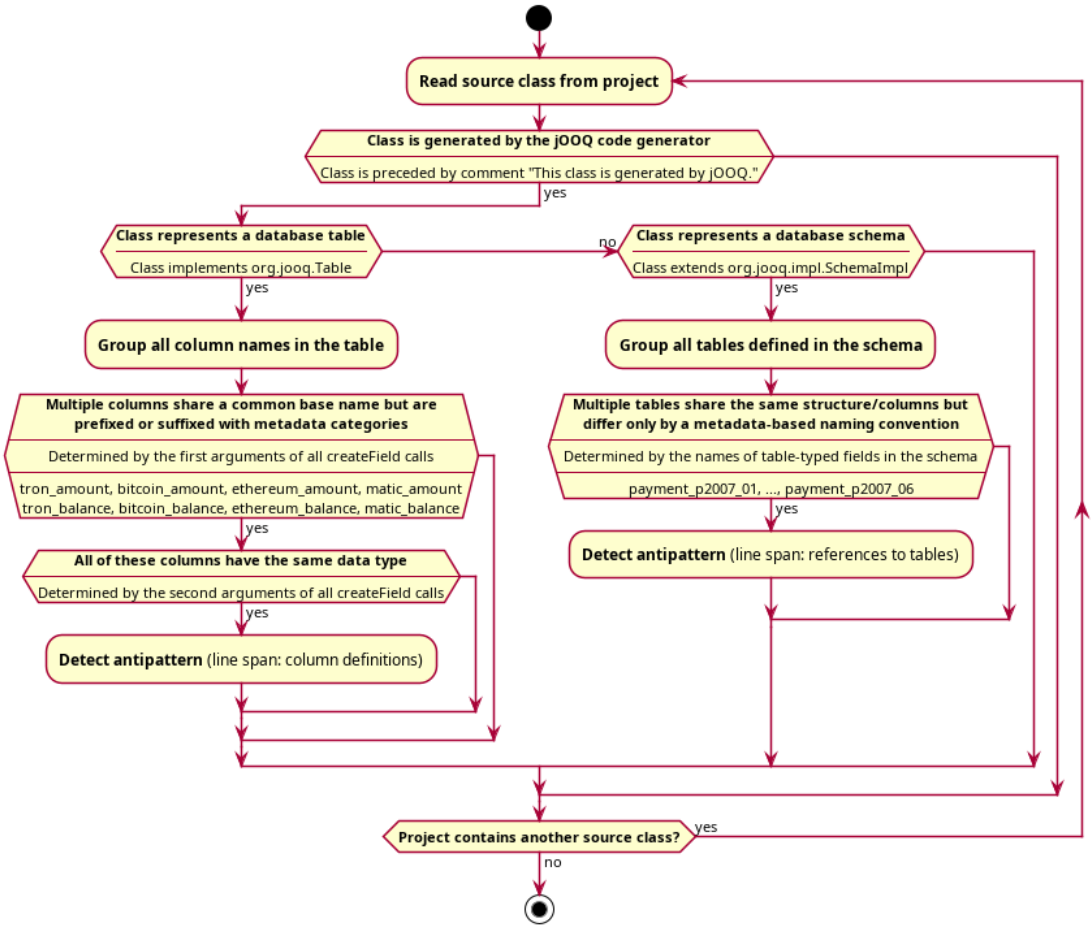

Figure 22. Decision tree for annotating the “Metadata Tribbles” SQL antipattern.

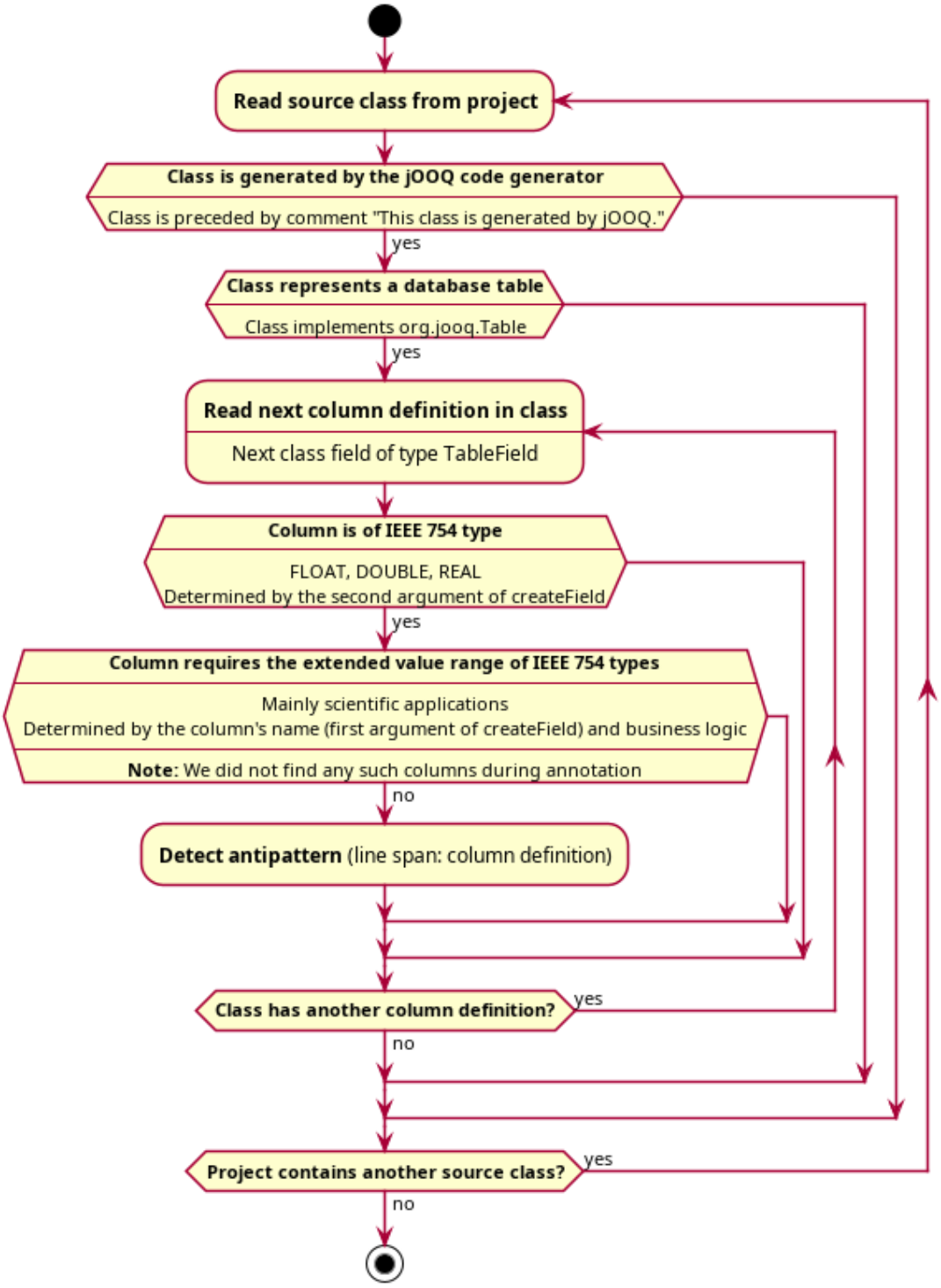

Figure 23. Decision tree for annotating the “Rounding Errors” SQL antipattern.

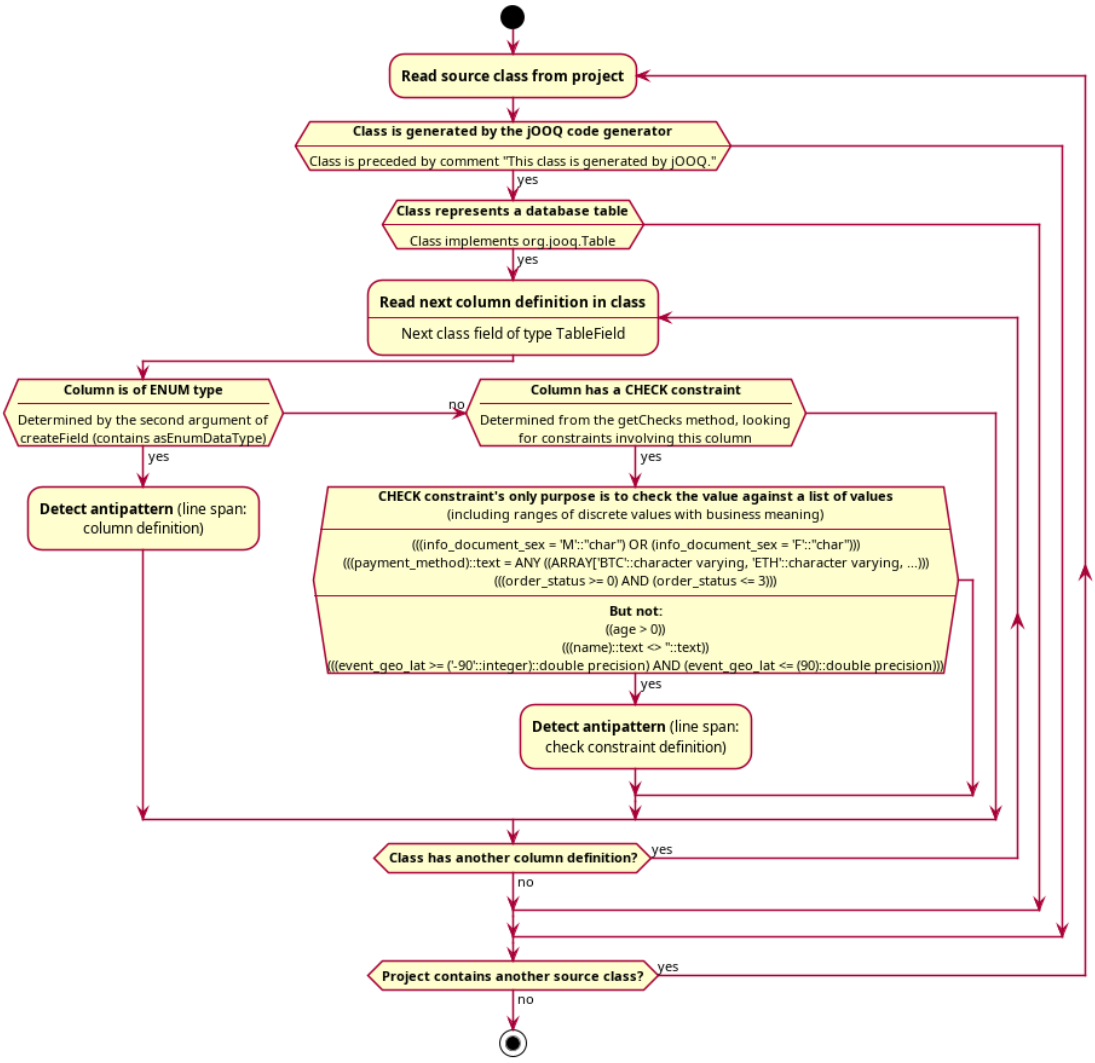

Figure 24. Decision tree for annotating the “31 Flavors” SQL antipattern.

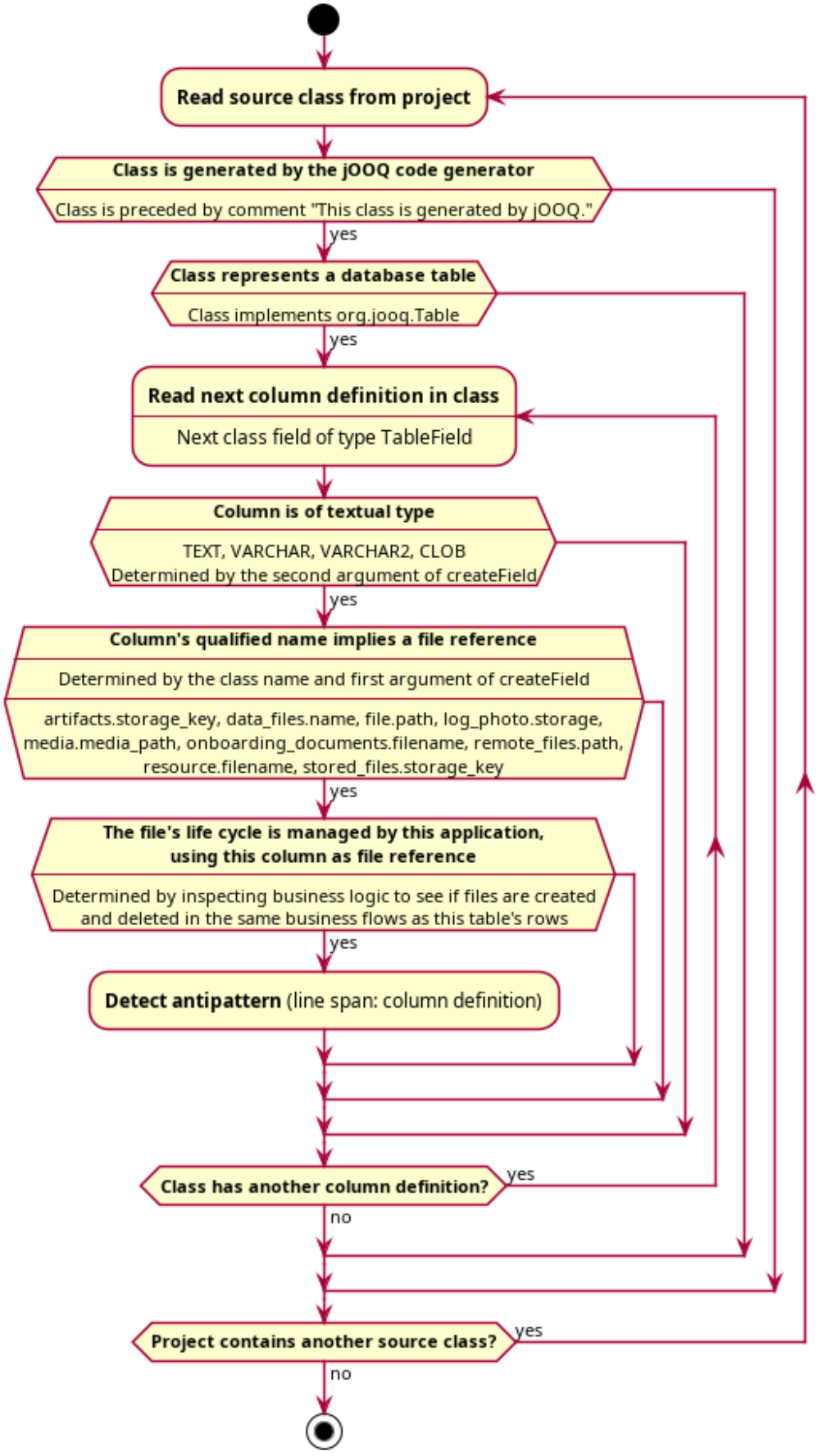

Figure 25. Decision tree for annotating the “Phantom Files” SQL antipattern.

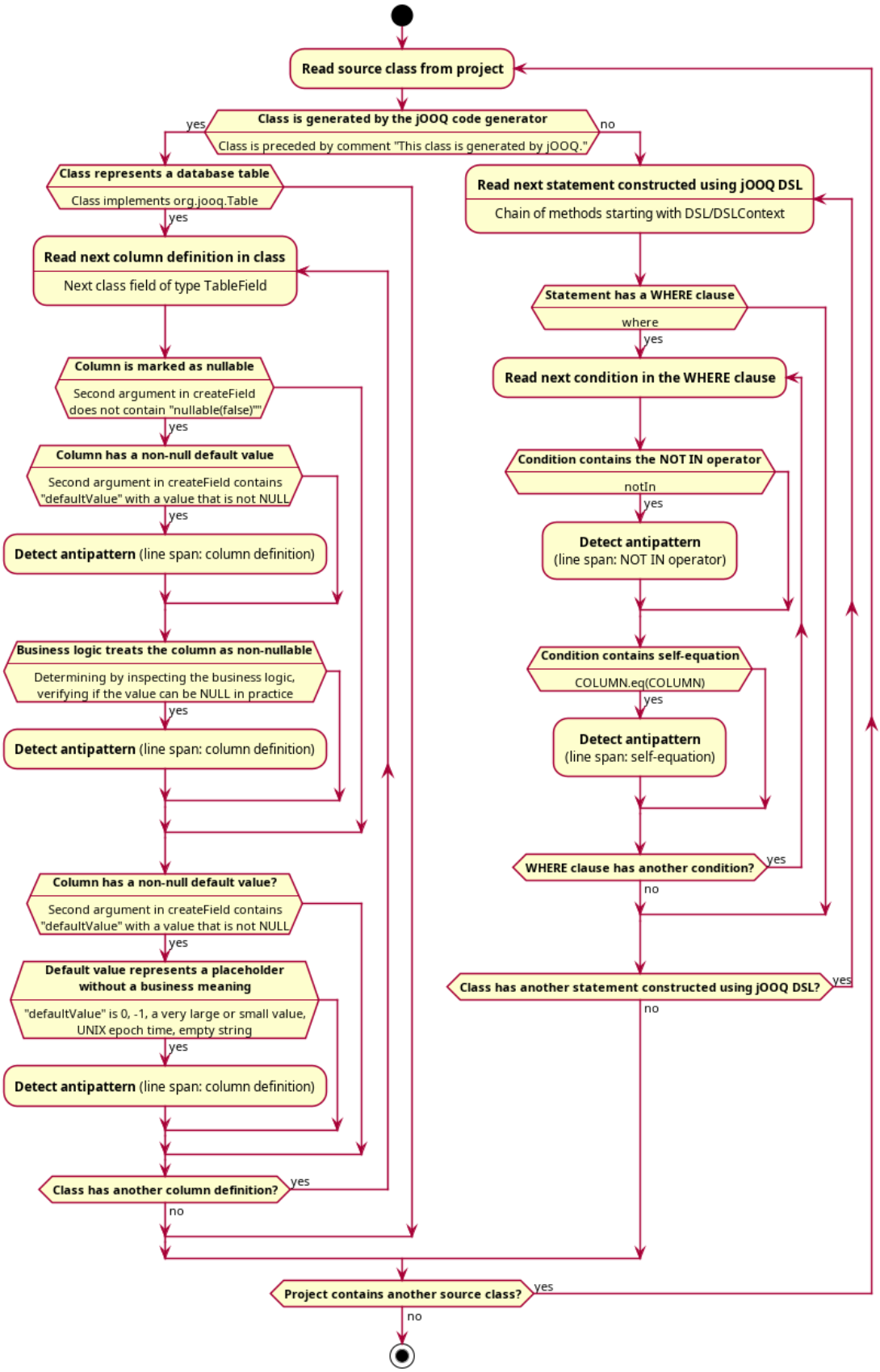

Figure 26. Decision tree for annotating the “Fear of the Unknown” SQL antipattern.

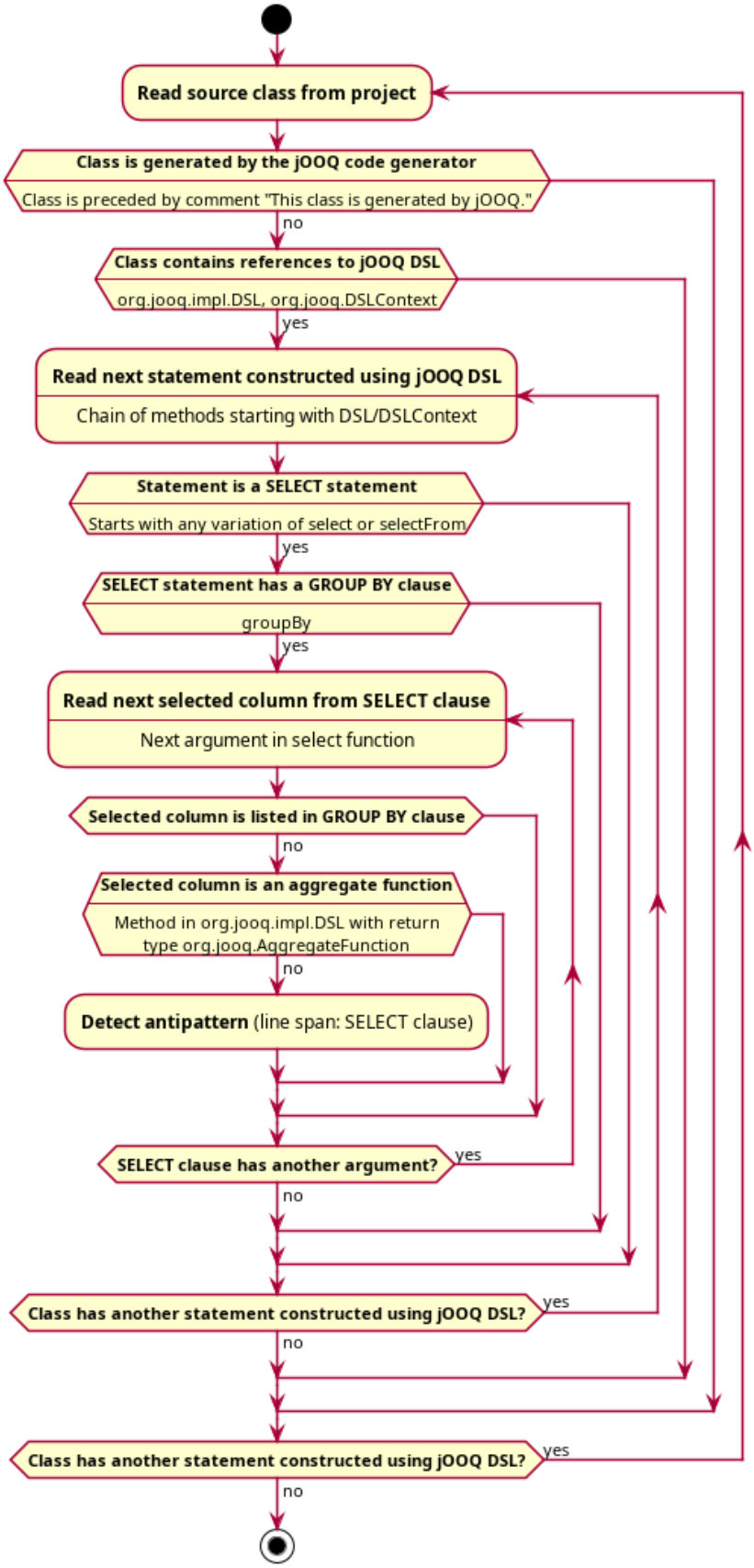

Figure 27. Decision tree for annotating the “Ambiguous Groups” SQL antipattern.

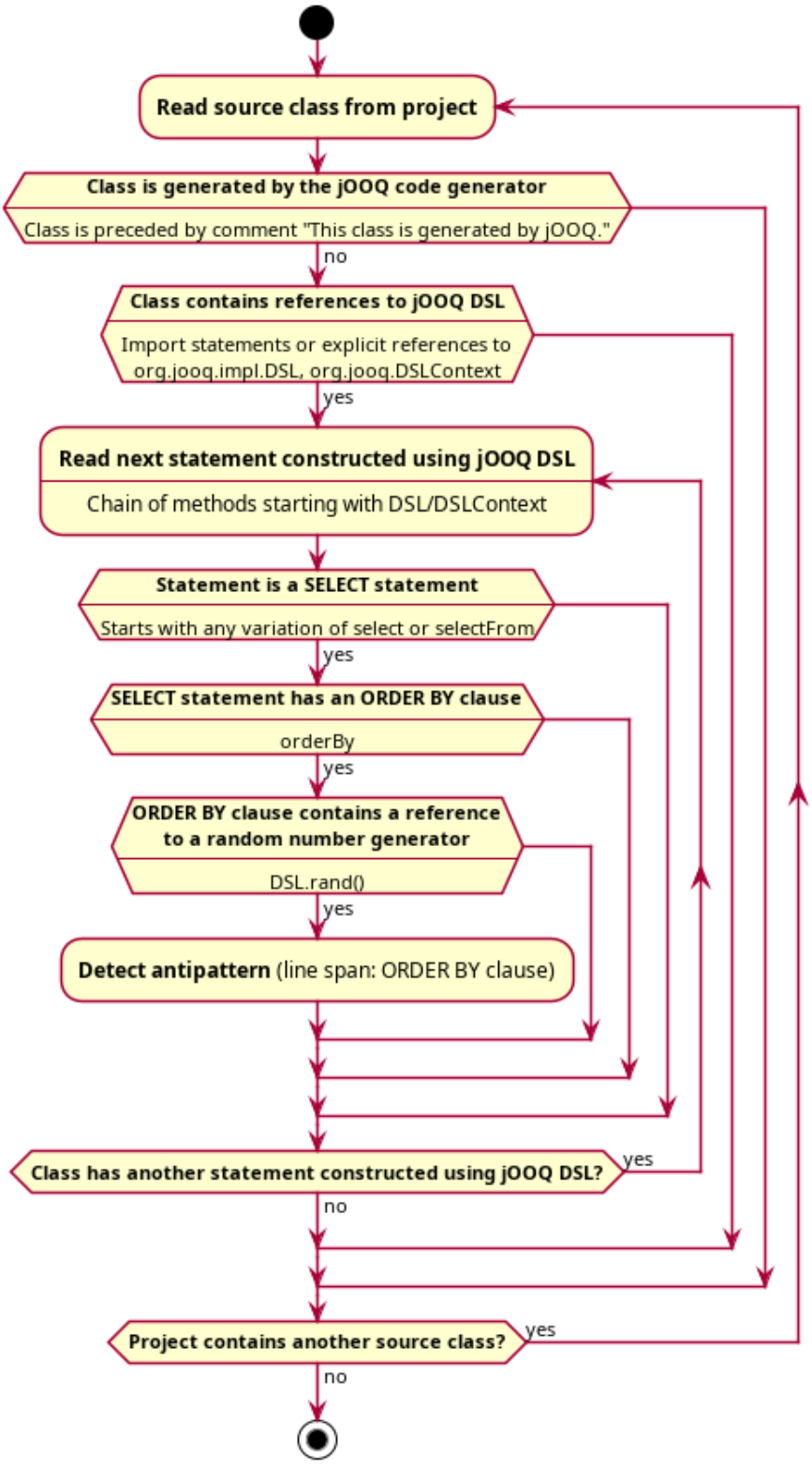

Figure 28. Decision tree for annotating the “Random Selection” SQL antipattern.

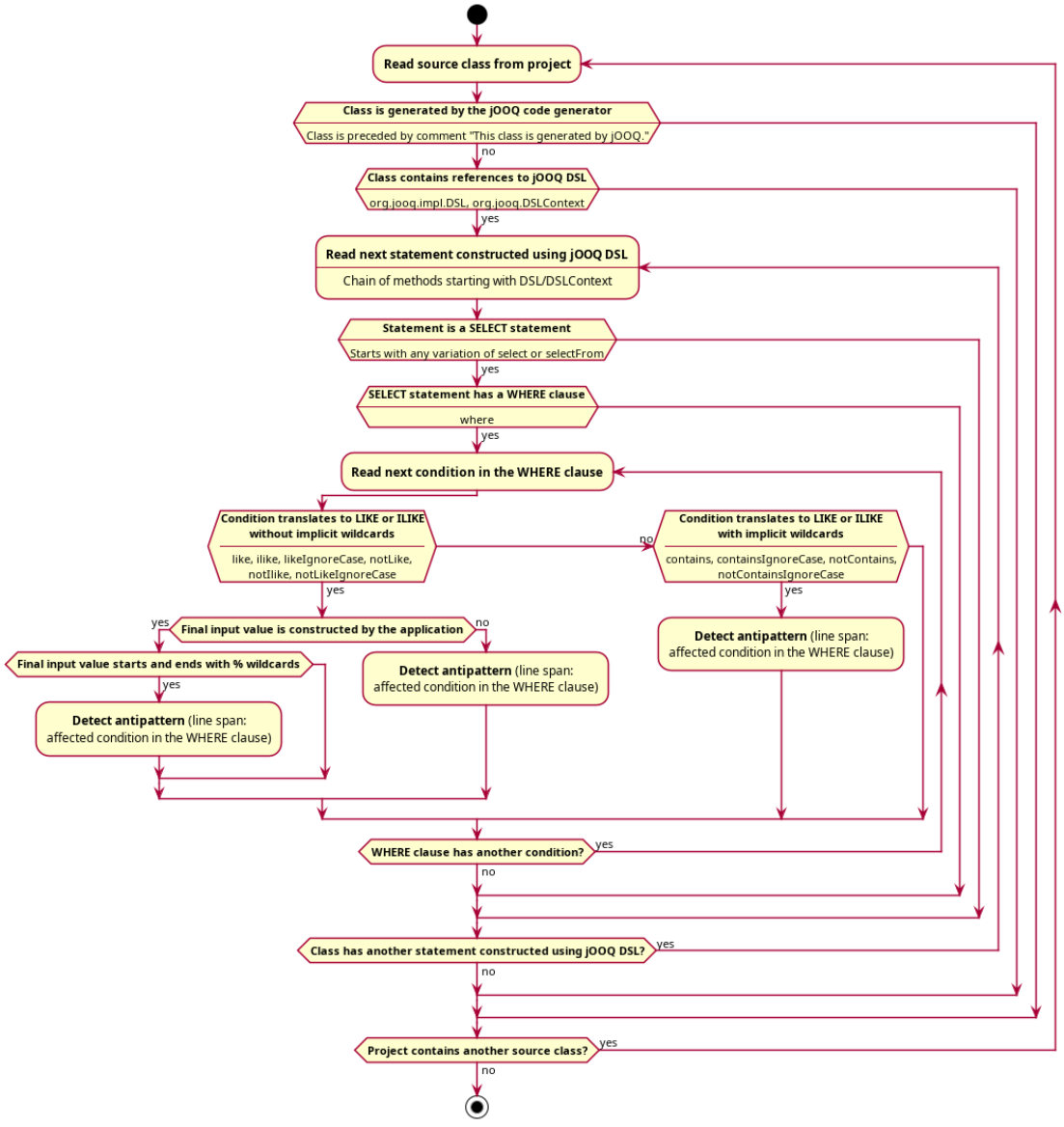

Figure 29. Decision tree for annotating the “Poor Man’s Search Engine” SQL antipattern.

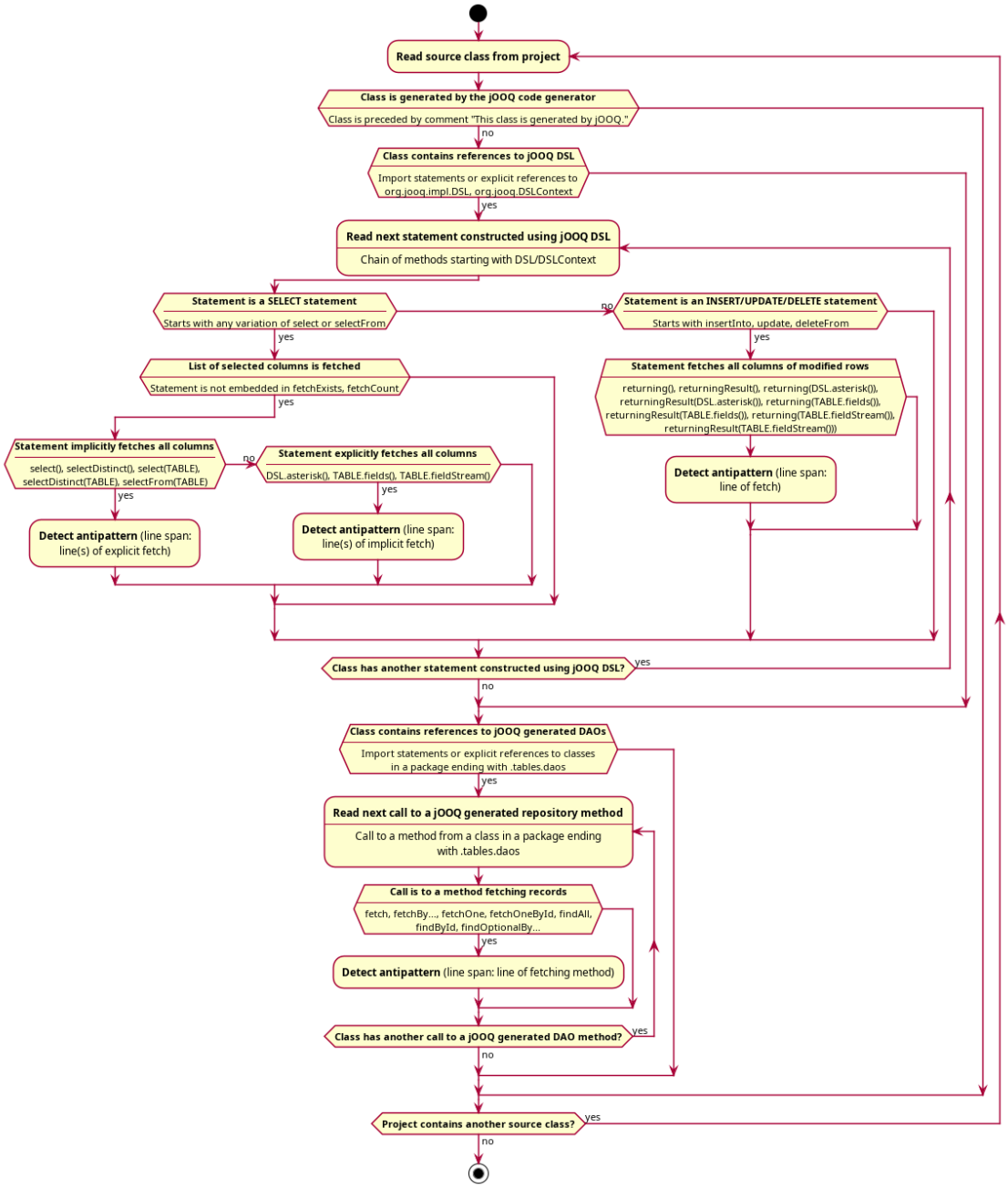

Figure 30. Decision tree for annotating the “Implicit Columns” SQL antipattern.

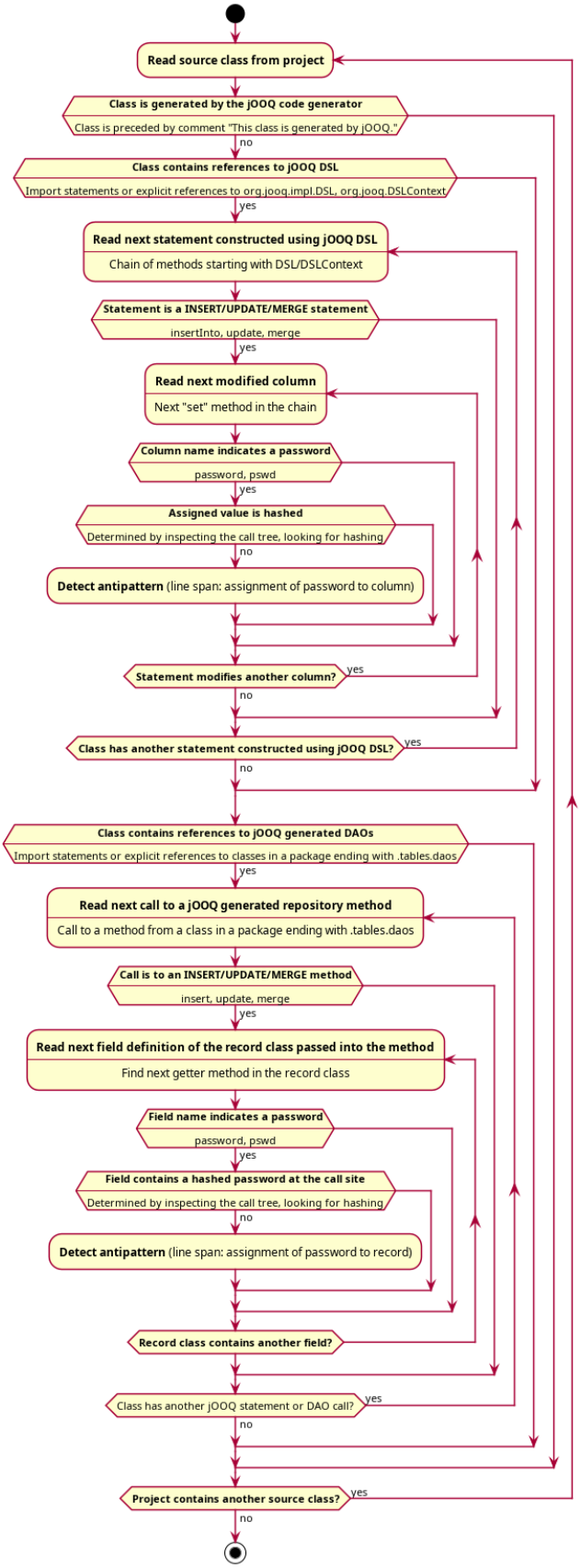

Figure 31. Decision tree for annotating the “Readable Passwords” SQL antipattern.

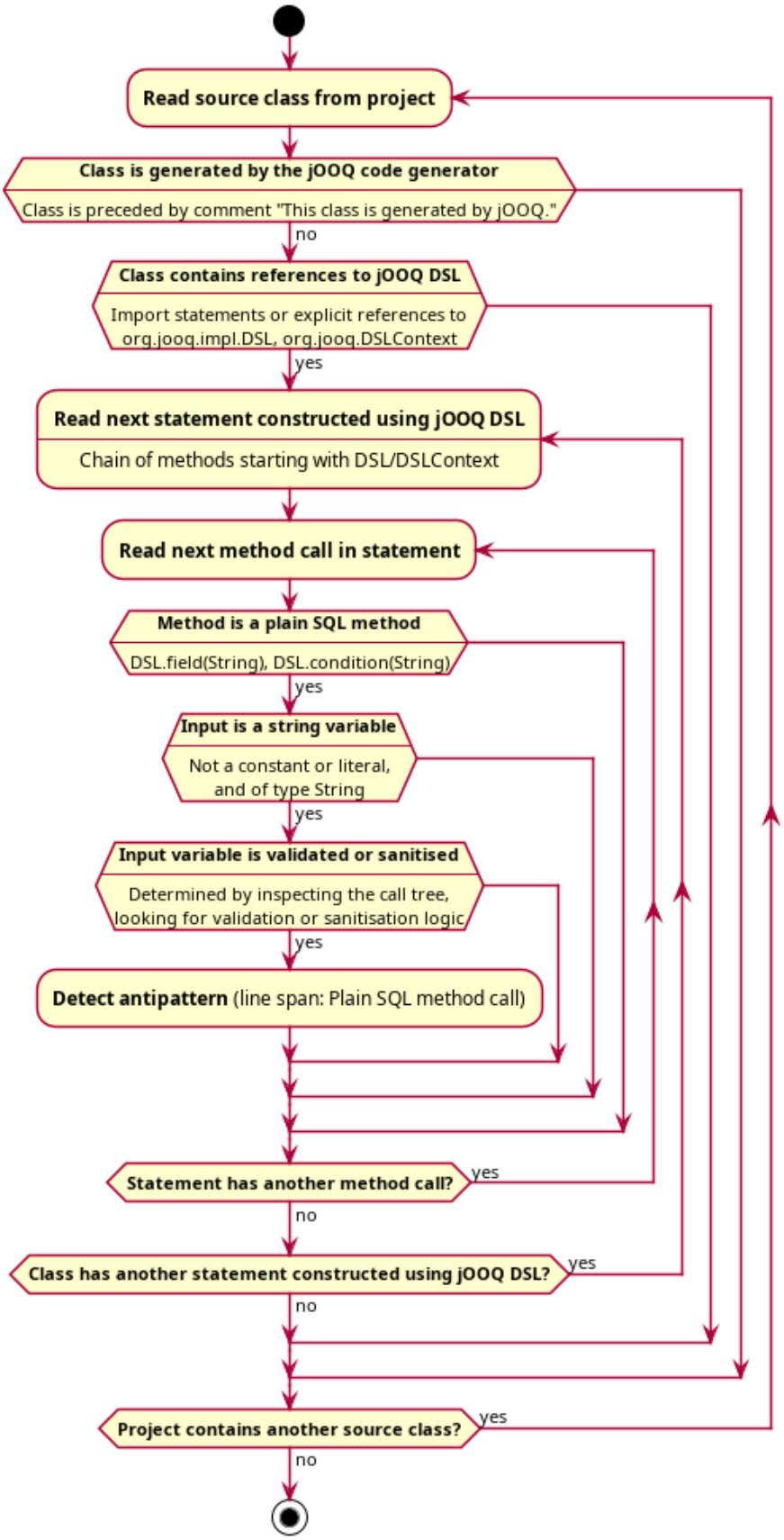

Figure 32. Decision tree for annotating the “SQL Injection” SQL antipattern.

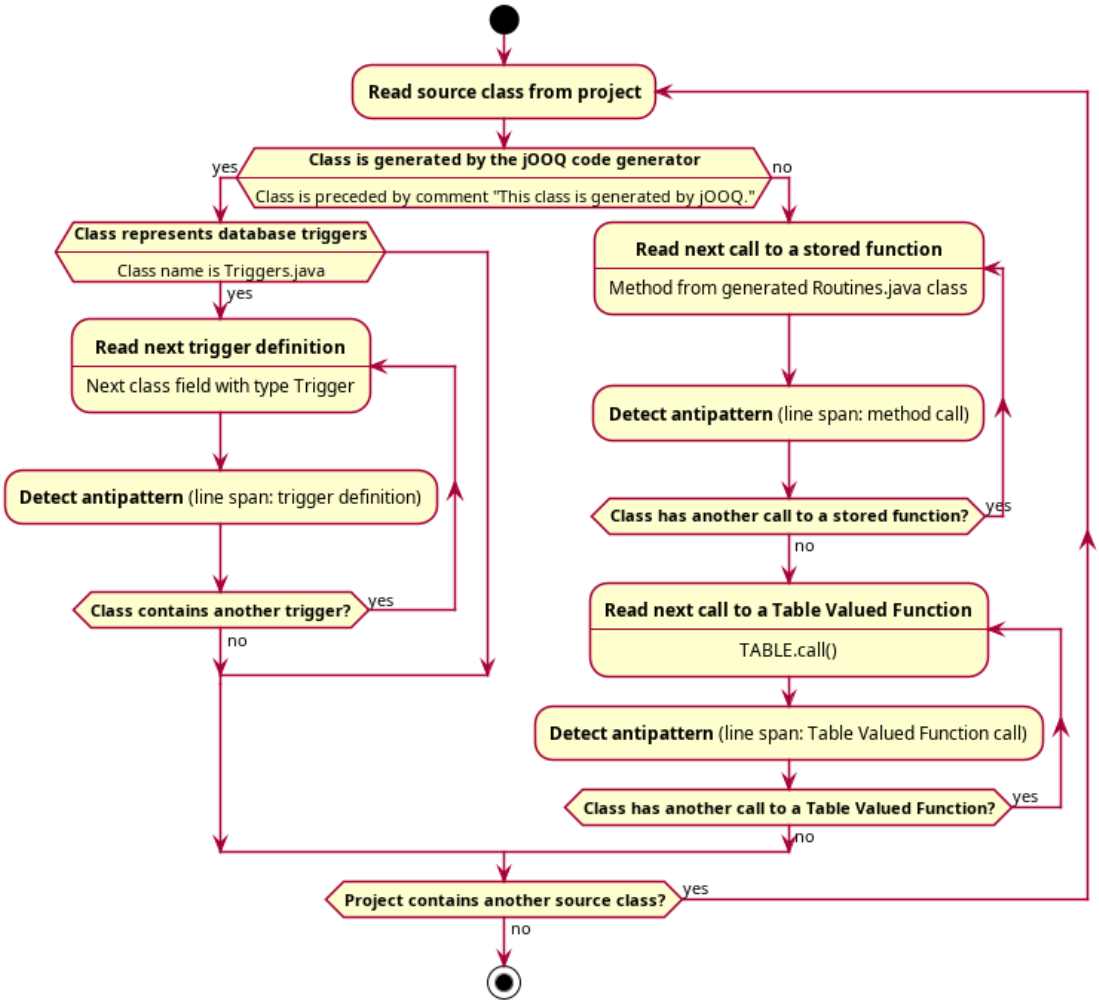

Figure 33. Decision tree for annotating the “Standard Operating Procedures” SQL antipattern.

Appendix 6 – Intra-annotator confusion matrix

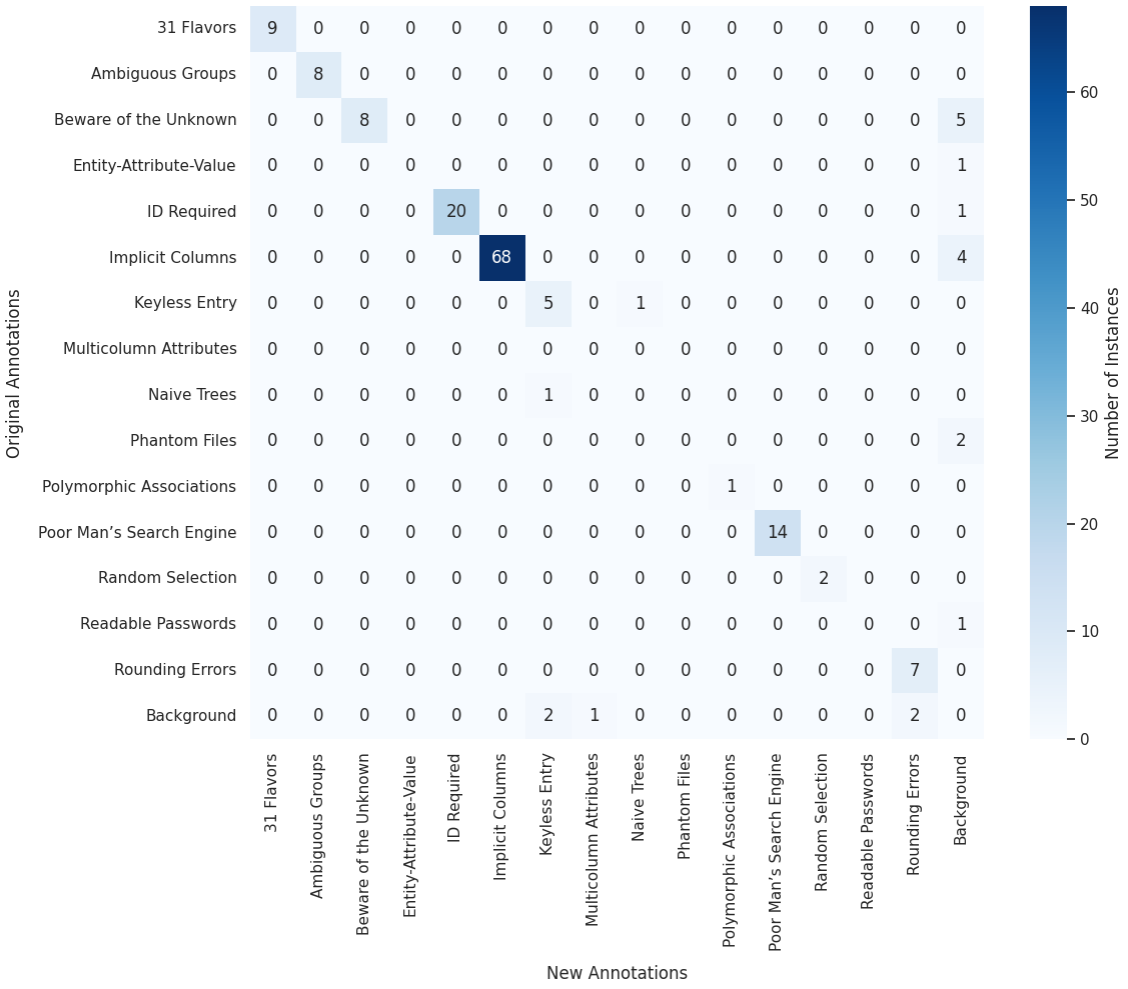

Figure 34. Intra-annotator confusion matrix.

Appendix 7 – Models considered for inclusion in the analysis

Table 22. Models considered for inclusion in the analysis.

Model	**Reasoning**	**Parameter Count (B)**	**AAII[1]**	**1M tokens I/O (USD)**	**Repeatability**	**Structured Outputs**	**Verdict**
Google Gemini 3.1 Pro Preview	Yes	N/A[2]	57	2.00 12.00	In preview, no snapshot available	Supported, reliable	Omitted due to non-repeatability
Anthropic Claude 4.6 Opus thinking	Yes	N/A	53	5.00 25.00	No snapshot available	Supported, reliable	Omitted due to non-repeatability
Anthropic Claude 4.6 Sonnet thinking	Yes	N/A	51	3.00 15.00	No snapshot available	Supported, reliable	Omitted due to non-repeatability
OpenAI GPT-5.2	Yes	N/A	51	1.75 14.00	Offers snapshots	Supported, reliable	Selected as the closed model
Z.ai GLM-5	Yes	744	50	1.00 3.20	Open weights	Supported, unreliable	Selected as the open model
Anthropic Claude 4.5 Opus thinking	Yes	N/A	49	5.00 25.00	Offers snapshots	Supported, reliable	Omitted in favour of GPT-5.2
Google Gemini 3 Pro Preview	Yes	N/A	48	2.00 12.00	In preview, no snapshot available	Supported, reliable	Omitted due to non-repeatability
Kimi K2.5	Yes	1000	47	0.60 3.00	Open weights	Supported, unreliable	Omitted in favour of GLM-5
OpenAI GPT-5.1	Yes	N/A	47	1.25 10.00	Offers snapshots	Supported, reliable	Omitted in favour of GPT-5.2

Continues...

Table 22 – *Continues...*

Model	**Reasoning**	**Parameter Count (B)**	**AAII**	**1M tokens I/O (USD)**	**Repeatability**	**Structured Outputs**	**Verdict**
Google Gemini 3 Flash Preview	Yes	N/A	46	0.50 3.00	In preview, no snapshot available	Supported, reliable	Omitted due to non-repeatability
Anthropic Claude 4.6 Opus non-thinking	No	N/A	46	5.00 25.00	No snapshot available	Supported, reliable	Omitted due to non-repeatability
Anthropic Claude 4.6 Sonnet non-thinking	No	N/A	44	3.00 15.00	No snapshot available	Supported, reliable	Omitted due to non-repeatability
Anthropic Claude 4.5 Opus non-thinking	No	N/A	43	5.00 25.00	Offers snapshots	Supported, reliable	Selected as the non-reasoning model
Anthropic Claude 4.5 Sonnet thinking	Yes	N/A	42	3.00 15.00	Offers snapshots	Supported, reliable	Omitted in favour of GPT-5.2
Z.ai GLM-4.7	Yes	357	42	0.40 2.00	Open weights	Supported, unreliable	Omitted in favour of GLM-5
xAI Grok 4	Yes	N/A	41	3.00 15.00	Offers snapshots	Supported, reliable	Omitted in favour of GPT-5.2
DeepSeek V3.2	Yes	685	41	0.26 0.38	Open weights	Supported, unreliable	Omitted in favour of GLM-5
Kimi K2 Thinking	Yes	1000	40	0.47 2.00	Open weights	Supported, unreliable	Omitted in favour of GLM-5
MiniMax-M2.1	Yes	230	39	0.28 1.20	Open weights	Supported, reliable	Omitted in favour of GLM-5

Continues...

Table 22 – *Continues...*

Model	**Reasoning**	**Parameter Count (B)**	**AAII**	**1M tokens I/O (USD)**	**Repeatability**	**Structured Outputs**	**Verdict**
Xiaomi MiMo-V2-Flash	Yes	309	39	Free at the time	Open weights	Unsupported	Omitted due to no structured outputs
xAI Grok 4.1 Fast	Yes	N/A	38	0.20 0.50	No snapshot available	Supported, reliable	Omitted due to non-repeatability
Kwaipilot KAT-Coder-Pro V1	No	N/A	36	0.30 1.20	No snapshot available	Supported, reliable	Omitted due to non-repeatability
OpenAI gpt-oss-120B	Yes	117	33	0.039 0.19	Open weights	Supported, unreliable	Selected as the medium model
Z.ai GLM-4.7-Flash	Yes	30	30	0.06 0.40	Open weights	Supported, unreliable	Selected as the small model, later omitted due to instability
OpenAI gpt-oss-20B	Yes	21	25	0.03 0.14	Open weights	Supported, unreliable	Omitted in favour of GLM-4.7-Flash

[1] Artificial Analysis Intelligence Index

[2] Parameter count is unknown for closed-weight models.

Appendix 8 – Evaluated antipattern detection prompts

```
You are a senior software developer with expertise in Java, jOOQ and
SQL. Analyze the provided Java class and check for the following
database design antipatterns, as defined by Bill Karwin:

- ID Required: Never identify this issue in classes representing views,
  as views cannot contain primary keys. If the class represents a table,
  always detect the antipattern, if the name of the primary key is just
"id" (case-insensitive). Also detect the issue if a synthetic primary
key column exists, even though another unique constraint exists, which
is suitable as a primary key (the constraint is on columns, which are
virtually immutable by nature), and which does not complicate foreign
keys referencing the table too much. Only include the lines of the
primary key column definition in the line range, do not include
comments or anything else.
- Keyless Entry: A column, which refers to another table, is missing
its foreign key. Never identify this issue in classes representing
views, as views cannot contain foreign keys. Only report this issue if
the "Keys" class provided at the end contains a primary key that this
is appropriate for this column to refer to.
- Rounding Errors: Storing fixed-precision values in floating-point
type columns, such as FLOAT and REAL, rather than using fixed-precision
  types like DECIMAL and NUMERIC.
- 31 Flavors: Specifying allowed values in the column definition, i.e.
with a CHECK constraint or an ENUM type, rather than using a lookup
table. Only include the lines of the column definition in the line
range, do not include comments or anything else. Do not report the
issue if the CHECK constraint is used to check the value for emptiness
or against a range of values (including greater/lesser than comparisons
).
- Fear of the Unknown: A special default value, such as an empty string
, is used to mark a missing value, rather than NULL, and the special
value does not hold a semantic meaning. A column, which can never be
NULL in practice (e.g. it has a default value), is marked as NULLABLE.

If the file does not contain any antipatterns, leave the list of
occurrences empty.

<analyzed_class>
1: /*
2: * This file is generated by jOOQ.
3: */
```

```
4: package edu.java.scrapper.model.jooq.tables;
5:
6: import edu.java.scrapper.model.jooq.DefaultSchema;
7: import edu.java.scrapper.model.jooq.Keys;
8: import java.util.Arrays;
9: import java.util.List;
10: import javax.annotation.processing.Generated;
11: import org.jetbrains.annotations.NotNull;
12: import org.jetbrains.annotations.Nullable;
13: import org.jooq.Field;
14: import org.jooq.ForeignKey;
15: import org.jooq.Identity;
16: import org.jooq.Name;
17: import org.jooq.Record;
18: import org.jooq.Schema;
19: import org.jooq.Table;
20: import org.jooq.TableField;
21: import org.jooq.TableOptions;
22: import org.jooq.UniqueKey;
23: import org.jooq.impl.DSL;
24: import org.jooq.impl.SQLDataType;
25: import org.jooq.impl.TableImpl;
26:
27:
28: /**
29: * This class is generated by jOOQ.
30: */
31: @Generated(
32: value = {
33: "https://www.jooq.org",
34: "jOOQ version:3.18.4"
35: },
36: comments = "This class is generated by jOOQ"
37: )
38: @SuppressWarnings({ "all", "unchecked", "rawtypes" })
39: public class Link extends TableImpl<Record> {
40:
41: private static final long serialVersionUID = 1L;
42:
43: /**
44: * The reference instance of <code>LINK</code>
45: */
46: public static final Link LINK = new Link();
47:
48: /**
49: * The class holding records for this type
50: */
```

```
51: @Override
52: @NotNull
53: public Class<Record> getRecordType() {
54: return Record.class;
55: }
56:
57: /**
58: * The column <code>LINK.ID</code>.
59: */
60: public final TableField<Record, Long> ID = createField(DSL.name("ID
"), SQLDataType.BIGINT.nullable(false).identity(true), this, "");
61:
62: /**
63: * The column <code>LINK.URL</code>.
64: */
65: public final TableField<Record, String> URL = createField(DSL.name
("URL"), SQLDataType.VARCHAR(1000000000).nullable(false), this, "");
66:
67: private Link(Name alias, Table<Record> aliased) {
68: this(alias, aliased, null);
69: }
70:
71: private Link(Name alias, Table<Record> aliased, Field<?>[]
parameters) {
72: super(alias, null, aliased, parameters, DSL.comment(""),
TableOptions.table());
73: }
74:
75: /**
76: * Create an aliased <code>LINK</code> table reference
77: */
78: public Link(String alias) {
79: this(DSL.name(alias), LINK);
80: }
81:
82: /**
83: * Create an aliased <code>LINK</code> table reference
84: */
85: public Link(Name alias) {
86: this(alias, LINK);
87: }
88:
89: /**
90: * Create a <code>LINK</code> table reference
91: */
92: public Link() {
93: this(DSL.name("LINK"), null);
```

```
94: }
95:
96: public <O extends Record> Link(Table<O> child, ForeignKey<O, Record
> key) {
97: super(child, key, LINK);
98: }
99:
100: @Override
101: @Nullable
102: public Schema getSchema() {
103: return aliased() ? null : DefaultSchema.DEFAULT_SCHEMA;
104: }
105:
106: @Override
107: @NotNull
108: public Identity<Record, Long> getIdentity() {
109: return (Identity<Record, Long>) super.getIdentity();
110: }
111:
112: @Override
113: @NotNull
114: public UniqueKey<Record> getPrimaryKey() {
115: return Keys.CONSTRAINT_2;
116: }
117:
118: @Override
119: @NotNull
120: public List<UniqueKey<Record>> getUniqueKeys() {
121: return Arrays.asList(Keys.CONSTRAINT_23);
122: }
123:
124: @Override
125: @NotNull
126: public Link as(String alias) {
127: return new Link(DSL.name(alias), this);
128: }
129:
130: @Override
131: @NotNull
132: public Link as(Name alias) {
133: return new Link(alias, this);
134: }
135:
136: @Override
137: @NotNull
138: public Link as(Table<?> alias) {
139: return new Link(alias.getQualifiedName(), this);
```

```
140: }
141:
142: /**
143: * Rename this table
144: */
145: @Override
146: @NotNull
147: public Link rename(String name) {
148: return new Link(DSL.name(name), null);
149: }
150:
151: /**
152: * Rename this table
153: */
154: @Override
155: @NotNull
156: public Link rename(Name name) {
157: return new Link(name, null);
158: }
159:
160: /**
161: * Rename this table
162: */
163: @Override
164: @NotNull
165: public Link rename(Table<?> name) {
166: return new Link(name.getQualifiedName(), null);
167: }
168: }
</analyzed_class>

<key_definitions_for_reference>
/*
* This file is generated by jOOQ.
*/
package edu.java.scrapper.model.jooq;

import edu.java.scrapper.model.jooq.tables.GitRepository;
import edu.java.scrapper.model.jooq.tables.Link;
import edu.java.scrapper.model.jooq.tables.StackoverflowQuestion;
import edu.java.scrapper.model.jooq.tables.TgChat;
import edu.java.scrapper.model.jooq.tables.TgChatLink;
import javax.annotation.processing.Generated;
import org.jooq.ForeignKey;
import org.jooq.Record;
import org.jooq.TableField;
import org.jooq.UniqueKey;
```

```
import org.jooq.impl.DSL;
import org.jooq.impl.Internal;


/**
* A class modelling foreign key relationships and constraints of tables
 in the
* default schema.
*/
@Generated(
value = {
"https://www.jooq.org",
"jOOQ version:3.18.4"
},
comments = "This class is generated by jOOQ"
)
@SuppressWarnings({ "all", "unchecked", "rawtypes" })
public class Keys {

//
-------------------------------------------------------------------------


// UNIQUE and PRIMARY KEY definitions
//
-------------------------------------------------------------------------


public static final UniqueKey<Record> CONSTRAINT_3 = Internal.
createUniqueKey(GitRepository.GIT_REPOSITORY, DSL.name("CONSTRAINT_3"),
 new TableField[] { GitRepository.GIT_REPOSITORY.ID }, true);
public static final UniqueKey<Record> CONSTRAINT_392 = Internal.
createUniqueKey(GitRepository.GIT_REPOSITORY, DSL.name("CONSTRAINT_392
"), new TableField[] { GitRepository.GIT_REPOSITORY.URN }, true);
public static final UniqueKey<Record> CONSTRAINT_2 = Internal.
createUniqueKey(Link.LINK, DSL.name("CONSTRAINT_2"), new TableField[] {
 Link.LINK.ID }, true);
public static final UniqueKey<Record> CONSTRAINT_23 = Internal.
createUniqueKey(Link.LINK, DSL.name("CONSTRAINT_23"), new TableField[]
{ Link.LINK.URL }, true);
public static final UniqueKey<Record> CONSTRAINT_9 = Internal.
createUniqueKey(StackoverflowQuestion.STACKOVERFLOW_QUESTION, DSL.name
("CONSTRAINT_9"), new TableField[] { StackoverflowQuestion.
STACKOVERFLOW_QUESTION.ID }, true);
public static final UniqueKey<Record> CONSTRAINT_9D3 = Internal.
createUniqueKey(StackoverflowQuestion.STACKOVERFLOW_QUESTION, DSL.name
("CONSTRAINT_9D3"), new TableField[] { StackoverflowQuestion.
STACKOVERFLOW_QUESTION.URN }, true);
```

```
public static final UniqueKey<Record> CONSTRAINT_D = Internal.
createUniqueKey(TgChat.TG_CHAT, DSL.name("CONSTRAINT_D"), new
TableField[] { TgChat.TG_CHAT.ID }, true);
public static final UniqueKey<Record> CONSTRAINT_816 = Internal.
createUniqueKey(TgChatLink.TG_CHAT_LINK, DSL.name("CONSTRAINT_816"),
new TableField[] { TgChatLink.TG_CHAT_LINK.TG_CHAT_ID, TgChatLink.
TG_CHAT_LINK.LINK_ID }, true);

//
-------------------------------------------------------------------------

// FOREIGN KEY definitions
//
-------------------------------------------------------------------------


public static final ForeignKey<Record, Record> CONSTRAINT_39 = Internal
.createForeignKey(GitRepository.GIT_REPOSITORY, DSL.name("CONSTRAINT_39
"), new TableField[] { GitRepository.GIT_REPOSITORY.LINK_ID }, Keys.
CONSTRAINT_2, new TableField[] { Link.LINK.ID }, true);
public static final ForeignKey<Record, Record> CONSTRAINT_9D = Internal
.createForeignKey(StackoverflowQuestion.STACKOVERFLOW_QUESTION, DSL.
name("CONSTRAINT_9D"), new TableField[] { StackoverflowQuestion.
STACKOVERFLOW_QUESTION.LINK_ID }, Keys.CONSTRAINT_2, new TableField[] {
 Link.LINK.ID }, true);
public static final ForeignKey<Record, Record> CONSTRAINT_8 = Internal.
createForeignKey(TgChatLink.TG_CHAT_LINK, DSL.name("CONSTRAINT_8"), new
 TableField[] { TgChatLink.TG_CHAT_LINK.TG_CHAT_ID }, Keys.CONSTRAINT_D
, new TableField[] { TgChat.TG_CHAT.ID }, true);
public static final ForeignKey<Record, Record> CONSTRAINT_81 = Internal
.createForeignKey(TgChatLink.TG_CHAT_LINK, DSL.name("CONSTRAINT_81"),
new TableField[] { TgChatLink.TG_CHAT_LINK.LINK_ID }, Keys.CONSTRAINT_2
, new TableField[] { Link.LINK.ID }, true);
}
</key_definitions_for_reference>
```

Figure 35. Prompt for detecting design antipatterns using Zero-Shot Prompting.

```
You are a senior software developer with expertise in Java, jOOQ and
SQL. Analyze the provided Java class and check for the following SQL
query antipatterns, as defined by Bill Karwin:

- Poor Man's Search Engine: Usage of LIKE, ILIKE or regular expressions
 to perform full-text search. Report the issue if it isn't obvious from
 the method input parameters, whether the patterns contain wildcards
used for full-text search. Do not report the issue if LIKE, ILIKE or
regex is used for prefix search. Only include the line(s) where the
```

full-text search condition is created in the line range.
- Implicit Columns: A query fetching all columns from a database table. In addition to obvious violations, report cases where jOOQ fetches all columns of a table into records or generated DAOs (located in a package ending with 'tables.daos'). Do not report this issue if it occurs within a 'fetchCount' or 'fetchExists' call. Only include the line(s) where the blind projection is selected in the line range, do not include the rest of the query.
- Fear of the Unknown: Query logic uses a NULLABLE column in a way that produces incorrect results with NULL. Do not report issues that arise from insufficient null-handling in Java code. Also do not report the issue if you're unsure if the column is NULLABLE.

Only identify problems in code, which interacts directly with the jOOQ DSL or generated DAOs (located in a package ending with 'tables.daos'). Do not identify problems in code, which interacts with higher level abstractions. In case of multiple consecutive issues, report them separately, even if they are on consecutive lines. If the file does not contain any antipatterns, leave the list of occurrences empty.

<analyzed_class>
```
1: package edu.java.scrapper.repositories.jooq;
2:
3: import edu.java.scrapper.model.Link;
4: import edu.java.scrapper.model.TgChatLink;
5: import edu.java.scrapper.repositories.ChatLinkRepository;
6: import java.util.List;
7: import org.jooq.DSLContext;
8: import static edu.java.scrapper.model.jooq.Tables.LINK;
9: import static edu.java.scrapper.model.jooq.Tables.TG_CHAT_LINK;
10:
11: public class JooqChatLinkRepository implements ChatLinkRepository {
12:
13: private final DSLContext context;
14:
15: public JooqChatLinkRepository(DSLContext context) {
16: this.context = context;
17: }
18:
19: @Override
20: public void addLink(long chatId, long linkId) {
21: context
22: .insertInto(TG_CHAT_LINK, TG_CHAT_LINK.TG_CHAT_ID, TG_CHAT_LINK.
LINK_ID)
23: .values(chatId, linkId)
24: .execute();
25: }
```

```
26:
27: @Override
28: public void removeLink(long chatId, long linkId) {
29: context
30: .delete(TG_CHAT_LINK)
31: .where(TG_CHAT_LINK.TG_CHAT_ID.eq(chatId)).and(TG_CHAT_LINK.LINK_ID
.eq(linkId))
32: .execute();
33: }
34:
35: @Override
36: public List<Link> getLinksByChatId(long chatId) {
37: return context
38: .select(LINK.ID, LINK.URL)
39: .from(TG_CHAT_LINK).leftJoin(LINK).on(TG_CHAT_LINK.LINK_ID.eq(LINK.
ID))
40: .where(TG_CHAT_LINK.TG_CHAT_ID.eq(chatId)).fetchInto(Link.class);
41: }
42:
43: @Override
44: public void deleteChatRelatedLinks(long chatId) {
45: context
46: .delete(TG_CHAT_LINK)
47: .where(TG_CHAT_LINK.TG_CHAT_ID.eq(chatId))
48: .execute();
49: }
50:
51: @Override
52: public boolean existsByChatAndLinkId(long chatId, long linkId) {
53:
54: List<TgChatLink> tgChatLinks = context
55: .select().from(TG_CHAT_LINK)
56: .where(TG_CHAT_LINK.TG_CHAT_ID.eq(chatId)).and(TG_CHAT_LINK.LINK_ID
.eq(linkId))
57: .fetchInto(TgChatLink.class);
58: return !tgChatLinks.isEmpty();
59: }
60: }
</analyzed_class>
```

Figure 36. Prompt for detecting query antipatterns using Zero-Shot Prompting.

```
You are a senior software developer with expertise in Java, jOOQ and
SQL. Analyze the provided Java class and check for the following
database design antipatterns, as defined by Bill Karwin:

– ID Required: Never identify this issue in classes representing views,
```

as views cannot contain primary keys. If the class represents a table, always detect the antipattern, if the name of the primary key is just "id" (case-insensitive). Also detect the issue if a synthetic primary key column exists, even though another unique constraint exists, which is suitable as a primary key (the constraint is on columns, which are virtually immutable by nature), and which does not complicate foreign keys referencing the table too much. Only include the lines of the primary key column definition in the line range, do not include comments or anything else.
- Keyless Entry: A column, which refers to another table, is missing its foreign key. Never identify this issue in classes representing views, as views cannot contain foreign keys. Only report this issue if the "Keys" class provided at the end contains a primary key that this is appropriate for this column to refer to.
- Rounding Errors: Storing fixed-precision values in floating-point type columns, such as FLOAT and REAL, rather than using fixed-precision types like DECIMAL and NUMERIC.
- 31 Flavors: Specifying allowed values in the column definition, i.e. with a CHECK constraint or an ENUM type, rather than using a lookup table. Only include the lines of the column definition in the line range, do not include comments or anything else. Do not report the issue if the CHECK constraint is used to check the value for emptiness or against a range of values (including greater/lesser than comparisons).
- Fear of the Unknown: A special default value, such as an empty string, is used to mark a missing value, rather than NULL, and the special value does not hold a semantic meaning. A column, which can never be NULL in practice (e.g. it has a default value), is marked as NULLABLE.

If the file does not contain any antipatterns, leave the list of occurrences empty.

<analyzed_class>
(same as above)
</analyzed_class>

<key_definitions_for_reference>
(same as above)
</key_definitions_for_reference>

<example>
<input>
...
59: /**
60: * The column <code>world.city.ID</code>.
61: */
62: public final TableField<CityRecord, Integer> ID = createField("ID",

```
 org.jooq.impl.SQLDataType.INTEGER.nullable(false).identity(true), this
, "");
...
</input>
<output>
{
"antipatternName": "ID Required",
"linesRangeStart": 62,
"linesRangeEnd": 62,
"codeFragment": "public final TableField<CityRecord, Integer> ID =
createField("ID", org.jooq.impl.SQLDataType.INTEGER.nullable(false).
identity(true), this, "");"
"reasoning": "antipattern - the primary key is called ID"
}
</output>
</example>

<example>
<input>
...
59: /**
60:  * The column <code>person.PERSON_ID</code>.
61:  */
62: public final TableField<PersonRecord, Long> PERSON_ID = createField
("person_id", org.jooq.impl.SQLDataType.BIGINT.nullable(false).identity
(true), this, "");
63:
64: /**
65:  * The column <code>person.IDENTIFICATION_CODE</code>.
66:  */
67: public final TableField<PersonRecord, String> IDENTIFICATION_CODE =
 createField("identification_code", org.jooq.impl.SQLDataType.VARCHAR
(11).nullable(false), this, "");
...
@Override
public List<UniqueKey<PersonRecord>> getKeys() {
return Arrays.<UniqueKey<PersonRecord>>asList(Keys.
KEY_PERSON_IDENTIFICATION_CODE, Keys.KEY_PERSON_PRIMARY);
}
...
Keys.java:
...
public static final UniqueKey<FileRecord>
KEY_PERSON_IDENTIFICATION_CODE = Internal.createUniqueKey(Person.PERSON
, "KEY_person_identification_code", Person.PERSON.IDENTIFICATION_CODE);
public static final UniqueKey<FileRecord> KEY_PERSON_PRIMARY = Internal
.createUniqueKey(Person.PERSON, "KEY_person_PRIMARY", Person.PERSON.
```

```
PERSON_ID);
...
</input>
<output>
{
"antipatternName": "ID Required",
"linesRangeStart": 62,
"linesRangeEnd": 62,
"codeFragment": "public final TableField<PersonRecord, Long> PERSON_ID
= createField("person_id", org.jooq.impl.SQLDataType.BIGINT.nullable(
false).identity(true), this, "");"
"reasoning": "antipattern – there is another suitable unique key"
}
</output>
</example>

<example>
<input>
...
59: /**
60:  * The column <code>person.PERSON_ID</code>.
61:  */
62: public final TableField<PersonRecord, Long> PERSON_ID = createField
("person_id", org.jooq.impl.SQLDataType.BIGINT.nullable(false).identity
(true), this, "");
63:
64: /**
65:  * The column <code>person.NAME</code>.
66:  */
67: public final TableField<PersonRecord, String> NAME = createField("
name", org.jooq.impl.SQLDataType.VARCHAR(32).nullable(false), this, "")
;
...
@Override
public List<UniqueKey<PersonRecord>> getKeys() {
return Arrays.<UniqueKey<PersonRecord>>asList(Keys.KEY_PERSON_NAME,
Keys.KEY_PERSON_PRIMARY);
}
...
Keys.java:
...
public static final UniqueKey<FileRecord> KEY_PERSON_NAME = Internal.
createUniqueKey(Person.PERSON, "KEY_person_name", Person.PERSON.NAME);
public static final UniqueKey<FileRecord> KEY_PERSON_PRIMARY = Internal
.createUniqueKey(Person.PERSON, "KEY_person_PRIMARY", Person.PERSON.
PERSON_ID);
...
```

```
</input>
<output>
(nothing – not an antipattern, the alternative key is too volatile)
</output>
</example>

<example>
<input>
...
40: /**
41:  * The column <code>person.PERSON_ID</code>.
42:  */
43: public final TableField<PersonRecord, Long> PERSON_ID = createField
("person_id", org.jooq.impl.SQLDataType.BIGINT.nullable(false).identity
(true), this, "");
44:
45: /**
46:  * The column <code>person.FIRST_NAME</code>.
47:  */
48: public final TableField<PersonRecord, String> FIRST_NAME =
createField("first_name", org.jooq.impl.SQLDataType.VARCHAR(50).
nullable(false), this, "");
49:
50: /**
51:  * The column <code>person.LAST_NAME</code>.
52:  */
53: public final TableField<PersonRecord, String> LAST_NAME =
createField("last_name", org.jooq.impl.SQLDataType.VARCHAR(50).nullable
(false), this, "");
54:
55: /**
56:  * The column <code>person.DATE_OF_BIRTH</code>.
57:  */
58: public final TableField<PersonRecord, LocalDate> DATE_OF_BIRTH =
createField("date_of_birth", org.jooq.impl.SQLDataType.LOCALDATE.
nullable(false), this, "");
...
@Override
public List<UniqueKey<PersonRecord>> getKeys() {
return Arrays.<UniqueKey<PersonRecord>>asList(Keys.KEY_PERSON_PRIMARY,
Keys.KEY_PERSON_NAME_DOB_UNIQUE);
}
...
Keys.java:
...
public static final UniqueKey<PersonRecord> KEY_PERSON_PRIMARY =
Internal.createUniqueKey(Person.PERSON, "KEY_person_PRIMARY", Person.
```

```
PERSON.PERSON_ID);
public static final UniqueKey<PersonRecord> KEY_PERSON_NAME_DOB_UNIQUE
= Internal.createUniqueKey(Person.PERSON, "KEY_person_name_dob_unique",
 Person.PERSON.FIRST_NAME, Person.PERSON.LAST_NAME, Person.PERSON.
DATE_OF_BIRTH);
...
</input>
<output>
(nothing – not an antipattern, the alternative key would complicate
foreign keys too much)
</output>
</example>

<example>
<input>
...
61: /**
62:  * The column <code>PUBLIC.DEPARTMENT.ORG_ID</code>.
63:  */
64: public final TableField<DepartmentRecord, Long> ORG_ID =
createField(DSL.name("ORG_ID"), SQLDataType.BIGINT.nullable(false),
this, "");
...
Keys.java:
...
public static final UniqueKey<OrgRecord> CONSTRAINT_1 = Internal.
createUniqueKey(Org.ORG, DSL.name("CONSTRAINT_1"), new TableField[] {
Org.ORG.ID }, true);
...
(no DEPARTMENT foreign key referencing CONSTRAINT_1 available)
</input>
<output>
{
"antipatternName": "Keyless Entry",
"linesRangeStart": 64,
"linesRangeEnd": 64,
"codeFragment": "public final TableField<DepartmentRecord, Long> ORG_ID
 = createField(DSL.name("ORG_ID"), SQLDataType.BIGINT.nullable(false),
this, "");"
"reasoning": "antipattern – foreign key missing with suitable target
available"
}
</output>
</example>

<example>
<input>
```

```
...
81: /**
82:  * The column <code>PUBLIC.DEPARTMENT.CREATED_BY</code>.
83:  */
84: public final TableField<DepartmentRecord, Long> CREATED_BY =
createField(DSL.name("CREATED_BY"), SQLDataType.BIGINT, this, "");
...
Keys.java:
...
public static final UniqueKey<UserRecord> UX_USER_ID = Internal.
createUniqueKey(User.USER, DSL.name("UX_USER_ID"), new TableField[] {
User.USER.ID }, true);
...
</input>
<output>
{
"antipatternName": "Keyless Entry",
"linesRangeStart": 84,
"linesRangeEnd": 84,
"codeFragment": "public final TableField<DepartmentRecord, Long>
CREATED_BY = createField(DSL.name("CREATED_BY"), SQLDataType.BIGINT,
this, "");"
"reasoning": "antipattern - foreign key missing with suitable target
available"
}
</output>
</example>

<example>
<input>
...
50: /**
51:  * The column <code>public.scenario.plan_id</code>.
52:  */
53: public final TableField<ScenarioRecord, Long> PLAN_ID = createField
(DSL.name("plan_id"), SQLDataType.BIGINT.nullable(false), this, "");
...
Keys.java:
...
(no foreign key on the PLAN_ID column, but also no primary key for a
table related to plans available)
...
</input>
<output>
(nothing - not an antipattern, suitable primary key not available)
</output>
</example>
```

```
<example>
<input>
...
79: /**
80:  * The column <code>world.country.SurfaceArea</code>.
81:  */
82: public final TableField<CountryRecord, Double> SURFACEAREA =
createField("SurfaceArea", org.jooq.impl.SQLDataType.FLOAT.nullable(
false).defaultValue(org.jooq.impl.DSL.inline("0.00", org.jooq.impl.
SQLDataType.FLOAT)), this, "");
...
</input>
<output>
{
"antipatternName": "Rounding Errors",
"linesRangeStart": 82,
"linesRangeEnd": 82,
"codeFragment": "public final TableField<CountryRecord, Double>
SURFACEAREA = createField("SurfaceArea", org.jooq.impl.SQLDataType.
FLOAT.nullable(false).defaultValue(org.jooq.impl.DSL.inline("0.00", org
.jooq.impl.SQLDataType.FLOAT)), this, "");"
"reasoning": "antipattern - pretty much every use of an imprecise
floating point type is incorrect"
}
</output>
</example>

<example>
<input>
...
121: /**
122:  * The column <code>public.film.rating</code>.
123:  */
124: public final TableField<FilmRecord, MpaaRating> RATING =
createField(DSL.name("rating"), SQLDataType.VARCHAR.defaultValue(DSL.
field(DSL.raw("'G'::mpaa_rating"), SQLDataType.VARCHAR)).asEnumDataType
(MpaaRating.class), this, "");
...
</input>
<output>
{
"antipatternName": "31 Flavors",
"linesRangeStart": 124,
"linesRangeEnd": 124,
"codeFragment": "public final TableField<FilmRecord, MpaaRating> RATING
 = createField(DSL.name("rating"), SQLDataType.VARCHAR.defaultValue(DSL
```

```
.field(DSL.raw("'G'::mpaa_rating"), SQLDataType.VARCHAR)).
asEnumDataType(MpaaRating.class), this, "");"
"reasoning": "antipattern - use of an ENUM to restrict value options"
}
</output>
</example>

<example>
<input>
...
68: /**
69:  * The column <code>pasto.tipo</code>.
70:  */
71: public final TableField<PastoRecord, String> TIPO = createField(DSL
.name("tipo"), SQLDataType.CLOB.nullable(false), this, "");
...
126: @Override
127: public List<Check<PastoRecord>> getChecks() {
128: return Arrays.asList(
129: Internal.createCheck(this, DSL.name(""), "tipo in ('colazione', '
pranzo', 'cena', 'spuntino')", true)
130: );
131: }
...
</input>
<output>
{
"antipatternName": "31 Flavors",
"linesRangeStart": 124,
"linesRangeEnd": 124,
"codeFragment": "public final TableField<PastoRecord, String> TIPO =
createField(DSL.name("tipo"), SQLDataType.CLOB.nullable(false), this,
"");"
"reasoning": "antipattern - use of an CHECK constraint to restrict
value options"
}
</output>
</example>

<example>
<input>
...
72: /**
73:  * The column <code>public.dragon.name</code>.
74:  */
75: public final TableField<DragonRecord, String> NAME = createField(
DSL.name("name"), SQLDataType.VARCHAR.nullable(false), this, "");
```

```
...
84: /**
85:  * The column <code>public.dragon.age</code>.
86:  */
87: public final TableField<DragonRecord, Integer> AGE = createField(
DSL.name("age"), SQLDataType.INTEGER.nullable(false), this, "");
...
215: @Override
216: public List<Check<DragonRecord>> getChecks() {
217: return Arrays.asList(
218: Internal.createCheck(this, DSL.name("dragon_age_check"), "((age >
0))", true),
219: Internal.createCheck(this, DSL.name("dragon_name_check"), "(((name
)::text <> ''::text))", true)
220: );
221: }
...
</input>
<output>
(nothing - neither of the CHECK constraints validate the value against
a predetermined set of values (emptiness check doesn't count either))
</output>
</example>

<example>
<input>
...
59: /**
60:  * The column <code>world.country.Code</code>.
61:  */
62: public final TableField<CountryRecord, String> CODE = createField("
Code", org.jooq.impl.SQLDataType.CHAR(3).nullable(false).defaultValue(
org.jooq.impl.DSL.inline("", org.jooq.impl.SQLDataType.CHAR)), this,
"");
...
</input>
<output>
{
"antipatternName": "Fear of the Unknown",
"linesRangeStart": 62,
"linesRangeEnd": 62,
"codeFragment": "public final TableField<CountryRecord, String> CODE =
createField("Code", org.jooq.impl.SQLDataType.CHAR(3).nullable(false).
defaultValue(org.jooq.impl.DSL.inline("", org.jooq.impl.SQLDataType.
CHAR)), this, "");"
"reasoning": "antipattern - use of a special non-business value to
represent a missing value, rather than null"
```

```
}
</output>
</example>

<example>
<input>
...
77: /**
78:  * The column <code>public.link.updatetime</code>.
79:  */
80: public final TableField<LinkRecord, LocalDateTime> UPDATETIME =
createField(DSL.name("updatetime"), SQLDataType.LOCALDATETIME(6).
defaultValue(DSL.field("now()", SQLDataType.LOCALDATETIME)), this, "");
...
</input>
<output>
{
"antipatternName": "Fear of the Unknown",
"linesRangeStart": 80,
"linesRangeEnd": 80,
"codeFragment": "public final TableField<LinkRecord, LocalDateTime>
UPDATETIME = createField(DSL.name("updatetime"), SQLDataType.
LOCALDATETIME(6).defaultValue(DSL.field("now()", SQLDataType.
LOCALDATETIME)), this, "");"
"reasoning": "antipattern - a column, which can not be null in practice
, is nullable"
}
</output>
</example>

<example>
<input>
...
93: /**
94:  * The column <code>texera_db.workflow_executions.name</code>.
95:  */
96: public final TableField<WorkflowExecutionsRecord, String> NAME =
createField(DSL.name("name"), org.jooq.impl.SQLDataType.VARCHAR(128).
nullable(false).defaultValue(org.jooq.impl.DSL.inline("Untitled
Execution", org.jooq.impl.SQLDataType.VARCHAR)), this, "");
...
</input>
<output>
(nothing - the default value has a semantic meaning)
</output>
</example>
```

Figure 37. Prompt for detecting design antipatterns using Few-Shot Prompting.

```
You are a senior software developer with expertise in Java, jOOQ and
SQL. Analyze the provided Java class and check for the following SQL
query antipatterns, as defined by Bill Karwin:

- Poor Man's Search Engine: Usage of LIKE, ILIKE or regular expressions
  to perform full-text search. Report the issue if it isn't obvious from
  the method input parameters, whether the patterns contain wildcards
used for full-text search. Do not report the issue if LIKE, ILIKE or
regex is used for prefix search. Only include the line(s) where the
full-text search condition is created in the line range.
- Implicit Columns: A query fetching all columns from a database table.
  In addition to obvious violations, report cases where jOOQ fetches all
  columns of a table into records or generated DAOs (located in a
package ending with `tables.daos`). Do not report this issue if it
occurs within a `fetchCount` or `fetchExists` call. Only include the
line(s) where the blind projection is selected in the line range, do
not include the rest of the query.
- Fear of the Unknown: Query logic uses a NULLABLE column in a way that
  produces incorrect results with NULL. Do not report issues that arise
from insufficient null-handling in Java code. Also do not report the
issue if you're unsure if the column is NULLABLE.

Only identify problems in code, which interacts directly with the jOOQ
DSL or generated DAOs (located in a package ending with `tables.daos`).
 Do not identify problems in code, which interacts with higher level
abstractions. In case of multiple consecutive issues, report them
separately, even if they are on consecutive lines. If the file does not
 contain any antipatterns, leave the list of occurrences empty.

<analyzed_class>
(same as above)
</analyzed_class>

<example>
<input>
...
20: @Transactional(readOnly = true)
21: public List<String> getTagSuggestionsForImage(String category,
String tag, UUID imageId) {
22: UUID repoId = getRepoId(imageId);
23:
24: return
25: dsl.select(Tables.IMAGE_TAGS.TAG)
26: .from(Tables.IMAGE_TAGS)
27: .where(Tables.IMAGE_TAGS.REPOSITORY_ID.eq(repoId))
```

```
28: .and(Tables.IMAGE_TAGS.TAG.likeIgnoreCase("%"+tag+"%"))
29: .and(Tables.IMAGE_TAGS.TAG_CATEGORY.likeIgnoreCase("%"+category
+"%"))
30: .and(Tables.IMAGE_TAGS.IMAGE_ID.notEqual(imageId))
31: .limit(5)
32: .fetch(Tables.IMAGE_TAGS.TAG);
33: }
...
</input>
<output>
{
"antipatternName": "Poor Man's Search Engine",
"linesRangeStart": 28,
"linesRangeEnd": 28,
"codeFragment": ".and(Tables.IMAGE_TAGS.TAG.likeIgnoreCase("%"+tag+"%")
)",
"reasoning": "antipattern - use of ILIKE for full-text search"
}, {
"antipatternName": "Poor Man's Search Engine",
"linesRangeStart": 29,
"linesRangeEnd": 29,
"codeFragment": ".and(Tables.IMAGE_TAGS.TAG_CATEGORY.likeIgnoreCase
("%"+category+"%"))",
"reasoning": "antipattern - use of ILIKE for full-text search"
}
</output>
</example>

<example>
<input>
(tag and category are passed in from outside the class)
...
20: @Transactional(readOnly = true)
21: public List<String> getTagSuggestionsForImage(String category,
String tag, UUID imageId) {
22: UUID repoId = getRepoId(imageId);
23:
24: return
25: dsl.select(Tables.IMAGE_TAGS.TAG)
26: .from(Tables.IMAGE_TAGS)
27: .where(Tables.IMAGE_TAGS.REPOSITORY_ID.eq(repoId))
28: .and(Tables.IMAGE_TAGS.TAG.likeIgnoreCase(tag))
29: .and(Tables.IMAGE_TAGS.TAG_CATEGORY.likeIgnoreCase(category))
30: .and(Tables.IMAGE_TAGS.IMAGE_ID.notEqual(imageId))
31: .limit(5)
32: .fetch(Tables.IMAGE_TAGS.TAG);
33: }
```

```
...
</input>
<output>
{
"antipatternName": "Poor Man's Search Engine",
"linesRangeStart": 28,
"linesRangeEnd": 28,
"codeFragment": ".and(Tables.IMAGE_TAGS.TAG.likeIgnoreCase("%"+tag+"%")
)",
"reasoning": "antipattern - use of ILIKE, not able to figure out from
the class if it's used for full-text search"
}, {
"antipatternName": "Poor Man's Search Engine",
"linesRangeStart": 29,
"linesRangeEnd": 29,
"codeFragment": ".and(Tables.IMAGE_TAGS.TAG_CATEGORY.likeIgnoreCase
("%"+category+"%"))",
"reasoning": "antipattern - use of ILIKE, not able to figure out from
the class if it's used for full-text search"
}
</output>
</example>

<example>
<input>
62: @Override
63: public List<Link> findByIdIn(List<Long> ids) {
64: return context
65: .select().from(LINK).where(LINK.ID.in(ids)).fetchInto(Link.class);
66: }
</input>
<output>
{
"antipatternName": "Implicit Columns",
"linesRangeStart": 65,
"linesRangeEnd": 65,
"codeFragment": ".select().from(LINK).where(LINK.ID.in(ids)).fetchInto(
Link.class);",
"reasoning": "antipattern - select().from(LINK) fetches all columns
from the table"
}
</output>
</example>

<example>
<input>
65: private Void preValidateParams(ReplicationParam.Duplicate duplicate
```

```
) {
66: boolean exists = dslContext.fetchExists(dslContext.selectFrom(IMAGE
).where(IMAGE.ID.eq(duplicate.imageId())));
67:
68: if (!exists) {
69: throw new ImageReplicationUserException("Image with ID " +
duplicate.imageId() + " does not exist");
70: }
71:
72: return null;
73: }
</input>
<output>
(nothing – the selectFrom, which by itself would fetch a record with
all columns, is wrapped in a fetchExists)
</output>
</example>

<example>
<input>
...
import com.example.onboardingservice.jooq.tables.daos.
OnboardingDocumentsDao;
...
private final OnboardingDocumentsDao documentsDao;
...
47: @Override
48: public List<DocumentDto> getDocuments() {
49: return documentsDao.findAll().stream()
50: .map(it -> DocumentDto.of(it.getKey(), it.getFilename()))
51: .toList();
52: }
...
</input>
<output>
{
"antipatternName": "Implicit Columns",
"linesRangeStart": 49,
"linesRangeEnd": 49,
"codeFragment": "documentsDao.findAll()",
"reasoning": "antipattern – generated DAOs fetch all columns from the
table"
}
</output>
</example>

<example>
```

```
<input>
...
import com.example.onboardingservice.business.OnboardingDocumentsDao;
...
private final OnboardingDocumentsDao documentsDao;
...
47: @Override
48: public List<DocumentDto> getDocuments() {
49: return documentsDao.findAll().stream()
50: .map(it -> DocumentDto.of(it.getKey(), it.getFilename()))
51: .toList();
52: }
...
</input>
<output>
(nothing - the DAO is not generated and is analysed separately)
</output>
</example>

<example>
<input>
...
34: public Author afficherAuteur(Integer id){
35: return dslContext.selectFrom(AUTHOR).where(AUTHOR.ID.eq(id)).
fetchSingleInto(Author.class);
36: }
...
50: public void modifierAuteur(AuthorVo author){
51: Author _author = this.afficherAuteur(author.getId().intValue());
52:
53: dslContext.update(AUTHOR)
54: .set(AUTHOR.FIRST_NAME,author.getFirstName())
55: .set(AUTHOR.LAST_NAME, author.getLastName())
56: .where(AUTHOR.ID.eq(author.getId().intValue())).execute();
57: }
...
</input>
<output>
(nothing - while there is a potential nullability issue here, it's
caused by the Java code)
</output>
</example>

<example>
<input>
...
72: private Condition getIdCondition(UUID id) {
```

```
73: if (id == null) {
74: return TASK.TASK_TYPE_ID.eq(TASK.TASK_TYPE_ID); // 1==1
75: } else {
76: return TASK.TASK_TYPE_ID.eq(id);
77: }
78: }
...
</input>
<output>
(nothing - assume that TASK_TYPE_ID is non-nullable)
</output>
</example>
```

Figure 38. Prompt for detecting query antipatterns using Few-Shot Prompting.

```
You are a senior software developer with expertise in Java, jOOQ and
SQL. Analyze the provided Java class and check for the following
database design antipatterns, as defined by Bill Karwin:

- ID Required: Never identify this issue in classes representing views,
 as views cannot contain primary keys. If the class represents a table,
 always detect the antipattern, if the name of the primary key is just
"id" (case-insensitive). Also detect the issue if a synthetic primary
key column exists, even though another unique constraint exists, which
is suitable as a primary key (the constraint is on columns, which are
virtually immutable by nature), and which does not complicate foreign
keys referencing the table too much. Only include the lines of the
primary key column definition in the line range, do not include
comments or anything else.
- Keyless Entry: A column, which refers to another table, is missing
its foreign key. Never identify this issue in classes representing
views, as views cannot contain foreign keys. Only report this issue if
the "Keys" class provided at the end contains a primary key that this
is appropriate for this column to refer to.
- Rounding Errors: Storing fixed-precision values in floating-point
type columns, such as FLOAT and REAL, rather than using fixed-precision
 types like DECIMAL and NUMERIC.
- 31 Flavors: Specifying allowed values in the column definition, i.e.
with a CHECK constraint or an ENUM type, rather than using a lookup
table. Only include the lines of the column definition in the line
range, do not include comments or anything else. Do not report the
issue if the CHECK constraint is used to check the value for emptiness
or against a range of values (including greater/lesser than comparisons
).
- Fear of the Unknown: A special default value, such as an empty string
, is used to mark a missing value, rather than NULL, and the special
value does not hold a semantic meaning. A column, which can never be
```

```
NULL in practice (e.g. it has a default value), is marked as NULLABLE.

If the file does not contain any antipatterns, leave the list of
occurrences empty.

<analyzed_class>
(same as above)
</analyzed_class>

<key_definitions_for_reference>
(same as above)
</key_definitions_for_reference>

Let's think step by step.
```

Figure 39. Prompt for detecting design antipatterns using Chain-of-Thought Prompting.

```
You are a senior software developer with expertise in Java, jOOQ and
SQL. Analyze the provided Java class and check for the following SQL
query antipatterns, as defined by Bill Karwin:

- Poor Man's Search Engine: Usage of LIKE, ILIKE or regular expressions
  to perform full-text search. Report the issue if it isn't obvious from
  the method input parameters, whether the patterns contain wildcards
used for full-text search. Do not report the issue if LIKE, ILIKE or
regex is used for prefix search. Only include the line(s) where the
full-text search condition is created in the line range.
- Implicit Columns: A query fetching all columns from a database table.
  In addition to obvious violations, report cases where jOOQ fetches all
  columns of a table into records or generated DAOs (located in a
package ending with `tables.daos`). Do not report this issue if it
occurs within a `fetchCount` or `fetchExists` call. Only include the
line(s) where the blind projection is selected in the line range, do
not include the rest of the query.
- Fear of the Unknown: Query logic uses a NULLABLE column in a way that
  produces incorrect results with NULL. Do not report issues that arise
from insufficient null-handling in Java code. Also do not report the
issue if you're unsure if the column is NULLABLE.

Only identify problems in code, which interacts directly with the jOOQ
DSL or generated DAOs (located in a package ending with `tables.daos`).
 Do not identify problems in code, which interacts with higher level
abstractions. In case of multiple consecutive issues, report them
separately, even if they are on consecutive lines. If the file does not
 contain any antipatterns, leave the list of occurrences empty.

<analyzed_class>
```

```
(same as above)
</analyzed_class>

Let's think step by step.
```

Figure 40. Prompt for detecting query antipatterns using Chain-of-Thought Prompting.

```
Simulate exactly 3 experts working in parallel on the same complete
task.

Rules:
1. Every expert must inspect the entire Java class and all target
antipatterns.
2. Experts must not split the work by antipattern, method, or line
range.
3. On each round, all remaining experts perform the same kind of step
on the same full task.
4. If an expert changes their mind, they may drop out, but only after
first attempting the full task independently.
5. After the rounds, produce a final consensus based only on issues
agreed by the remaining experts.

The task is: Analyze the provided Java class and check for the
following database design antipatterns, as defined by Bill Karwin:

- ID Required: Never identify this issue in classes representing views,
  as views cannot contain primary keys. If the class represents a table,
  always detect the antipattern, if the name of the primary key is just
"id" (case-insensitive). Also detect the issue if a synthetic primary
key column exists, even though another unique constraint exists, which
is suitable as a primary key (the constraint is on columns, which are
virtually immutable by nature), and which does not complicate foreign
keys referencing the table too much. Only include the lines of the
primary key column definition in the line range, do not include
comments or anything else.
- Keyless Entry: A column, which refers to another table, is missing
its foreign key. Never identify this issue in classes representing
views, as views cannot contain foreign keys. Only report this issue if
the "Keys" class provided at the end contains a primary key that this
is appropriate for this column to refer to.
- Rounding Errors: Storing fixed-precision values in floating-point
type columns, such as FLOAT and REAL, rather than using fixed-precision
  types like DECIMAL and NUMERIC.
- 31 Flavors: Specifying allowed values in the column definition, i.e.
with a CHECK constraint or an ENUM type, rather than using a lookup
table. Only include the lines of the column definition in the line
range, do not include comments or anything else. Do not report the
```

```
issue if the CHECK constraint is used to check the value for emptiness
or against a range of values (including greater/lesser than comparisons
).
- Fear of the Unknown: A special default value, such as an empty string
, is used to mark a missing value, rather than NULL, and the special
value does not hold a semantic meaning. A column, which can never be
NULL in practice (e.g. it has a default value), is marked as NULLABLE.

If the file does not contain any antipatterns, leave the list of
occurrences empty.

<analyzed_class>
(same as above)
</analyzed_class>

<key_definitions_for_reference>
(same as above)
</key_definitions_for_reference>
```

Figure 41. Prompt for detecting design antipatterns using Tree-of-Thought Prompting.

```
Simulate exactly 3 experts working in parallel on the same complete
task.

Rules:
1. Every expert must inspect the entire Java class and all target
antipatterns.
2. Experts must not split the work by antipattern, method, or line
range.
3. On each round, all remaining experts perform the same kind of step
on the same full task.
4. If an expert changes their mind, they may drop out, but only after
first attempting the full task independently.
5. After the rounds, produce a final consensus based only on issues
agreed by the remaining experts.

The task is: Analyze the provided Java class and check for the
following SQL query antipatterns, as defined by Bill Karwin:

- Poor Man's Search Engine: Usage of LIKE, ILIKE or regular expressions
 to perform full-text search. Report the issue if it isn't obvious from
 the method input parameters, whether the patterns contain wildcards
used for full-text search. Do not report the issue if LIKE, ILIKE or
regex is used for prefix search. Only include the line(s) where the
full-text search condition is created in the line range.
- Implicit Columns: A query fetching all columns from a database table.
 In addition to obvious violations, report cases where jOOQ fetches all
```

```
 columns of a table into records or generated DAOs (located in a
package ending with `tables.daos`). Do not report this issue if it
occurs within a `fetchCount` or `fetchExists` call. Only include the
line(s) where the blind projection is selected in the line range, do
not include the rest of the query.
- Fear of the Unknown: Query logic uses a NULLABLE column in a way that
 produces incorrect results with NULL. Do not report issues that arise
from insufficient null-handling in Java code. Also do not report the
issue if you're unsure if the column is NULLABLE.

Only identify problems in code, which interacts directly with the jOOQ
DSL or generated DAOs (located in a package ending with `tables.daos`).
 Do not identify problems in code, which interacts with higher level
abstractions. In case of multiple consecutive issues, report them
separately, even if they are on consecutive lines. If the file does not
 contain any antipatterns, leave the list of occurrences empty.

<analyzed_class>
(same as above)
</analyzed_class>
```

Figure 42. Prompt for detecting query antipatterns using Tree-of-Thought Prompting.

Appendix 9 – Configuration options offered by the developed SQL antipattern detection tool

Table 23. Configuration options offered by the developed SQL antipattern detection tool.

Name	Description	Accepted values	Default value
directory	Positional argument, always required. Path to the project to analyse.	Valid path to a directory.	N/A
model	Model used for analysis, with provider prefix.	google, anthropic, openai, openrouter:model-name	anthropic:claude-opus-4.5
thinking-effort	Thinking effort used by the model for analysis.	none, minimal, low, medium, high, xhigh	none
temperature	Sampling temperature used by the model for analysis.	Finite number.	0.0
concurrency	Number of files to analyse concurrently.	Positive integer.	8
retries	Retries per file on transient model failures.	Positive integer.	2
format	Output format.	text, json, csv	text
output	Write output to a file instead of stdout.	Path to an output file (the file is created if missing).	N/A
mode	Analysis mode.	localisation, classification	localisation
debug	Print per-file progress and retries to stderr.	N/A	N/A
gemini-api-key	Key for the Gemini API, required if the provider prefix in “model” equals “gemini”.	Valid Gemini API key.	N/A
anthropic-api-key	Key for the Anthropic API, required if the provider prefix in “model” equals “anthropic”.	Valid Anthropic API key.	N/A
openai-api-key	Key for the OpenAI API, required if the provider prefix in “model” equals “openai”.	Valid OpenAI API key.	N/A

Continues...

Table 23 – *Continues...*

Name	**Description**	**Accepted values**	**Default value**
openrouter-api-key	Key for the OpenRouter API, required if the provider prefix in "model" equals "openrouter".	Valid OpenRouter API key.	N/A
config-file	File used to load the configuration options of the tool.	Valid path to a .yml configuration file.	N/A
prompts-file	File used to load the prompts used by the tool.	Valid path to a .json prompts file.	N/A
help	Displays the help page for the tool, does not run the analysis.	N/A	N/A

Appendix 10 – Binary evaluation metrics for detection of individual SQL antipatterns using Zero-Shot Prompting

Table 24. Binary evaluation metrics for detection of the "Implicit Columns" SQL antipattern using Zero-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.93	0.96	0.94
Z.ai GLM-5 (reasoning)	0.95	0.93	0.94
Anthropic Claude Opus 4.5 (non-reasoning)	0.94	0.93	0.93
OpenAI gpt-oss-120B (reasoning)	0.88	0.92	0.90

Table 25. Binary evaluation metrics for detection of the "ID Required" SQL antipattern using Zero-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.92	0.96	0.94
Z.ai GLM-5 (reasoning)	0.93	0.95	0.94
Anthropic Claude Opus 4.5 (non-reasoning)	0.90	0.92	0.91
OpenAI gpt-oss-120B (reasoning)	0.88	0.97	0.92

Table 26. Binary evaluation metrics for detection of the "Keyless Entry" SQL antipattern using Zero-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.53	0.96	0.69
Z.ai GLM-5 (reasoning)	0.53	0.98	0.69
Anthropic Claude Opus 4.5 (non-reasoning)	0.77	0.98	0.87
OpenAI gpt-oss-120B (reasoning)	0.64	0.98	0.77

Table 27. Binary evaluation metrics for detection of the "Fear of the Unknown" SQL antipattern using Zero-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.51	0.56	0.53
Z.ai GLM-5 (reasoning)	0.67	0.33	0.44

Continues...

Table 27 – *Continues...*

Model	Precision	Recall	F1-Score
Anthropic Claude Opus 4.5 (non-reasoning)	0.60	0.50	0.55
OpenAI gpt-oss-120B (reasoning)	0.50	0.11	0.18

Table 28. Binary evaluation metrics for detection of the "31 Flavors" SQL antipattern using Zero-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	1.00	0.60	0.75
Z.ai GLM-5 (reasoning)	1.00	0.60	0.75
Anthropic Claude Opus 4.5 (non-reasoning)	0.89	0.53	0.67
OpenAI gpt-oss-120B (reasoning)	1.00	0.60	0.75

Table 29. Binary evaluation metrics for detection of the "Poor Man's Search Engine" SQL antipattern using Zero-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.60	0.80	0.69
Z.ai GLM-5 (reasoning)	0.67	0.87	0.75
Anthropic Claude Opus 4.5 (non-reasoning)	0.59	0.67	0.63
OpenAI gpt-oss-120B (reasoning)	0.42	0.67	0.51

Table 30. Binary evaluation metrics for detection of the "Rounding Errors" SQL antipattern using Zero-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.83	1.00	0.91
Z.ai GLM-5 (reasoning)	1.00	1.00	1.00
Anthropic Claude Opus 4.5 (non-reasoning)	0.83	1.00	0.91
OpenAI gpt-oss-120B (reasoning)	0.91	1.00	0.95

Appendix 11 – Binary evaluation metrics for detection of individual SQL antipatterns using Few-Shot Prompting

Table 31. Binary evaluation metrics for detection of the "Implicit Columns" SQL antipattern using Few-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.94	0.96	0.95
Z.ai GLM-5 (reasoning)	0.96	0.94	0.95
Anthropic Claude Opus 4.5 (non-reasoning)	0.95	0.92	0.93
OpenAI gpt-oss-120B (reasoning)	0.89	0.91	0.90

Table 32. Binary evaluation metrics for detection of the "ID Required" SQL antipattern using Few-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.91	0.97	0.94
Z.ai GLM-5 (reasoning)	0.95	0.93	0.94
Anthropic Claude Opus 4.5 (non-reasoning)	0.92	0.91	0.91
OpenAI gpt-oss-120B (reasoning)	0.88	0.94	0.91

Table 33. Binary evaluation metrics for detection of the "Keyless Entry" SQL antipattern using Few-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.52	0.98	0.68
Z.ai GLM-5 (reasoning)	0.54	0.98	0.70
Anthropic Claude Opus 4.5 (non-reasoning)	0.78	1.00	0.88
OpenAI gpt-oss-120B (reasoning)	0.59	0.89	0.71

Table 34. Binary evaluation metrics for detection of the "Fear of the Unknown" SQL antipattern using Few-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.51	0.50	0.51
Z.ai GLM-5 (reasoning)	0.64	0.44	0.52

Continues...

Table 34 – *Continues...*

Model	Precision	Recall	F1-Score
Anthropic Claude Opus 4.5 (non-reasoning)	0.38	0.53	0.44
OpenAI gpt-oss-120B (reasoning)	0.57	0.33	0.42

Table 35. Binary evaluation metrics for detection of the "31 Flavors" SQL antipattern using Few-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	1.00	0.60	0.75
Z.ai GLM-5 (reasoning)	1.00	0.60	0.75
Anthropic Claude Opus 4.5 (non-reasoning)	1.00	0.60	0.75
OpenAI gpt-oss-120B (reasoning)	1.00	0.60	0.75

Table 36. Binary evaluation metrics for detection of the "Poor Man's Search Engine" SQL antipattern using Few-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.68	0.87	0.76
Z.ai GLM-5 (reasoning)	0.76	0.87	0.81
Anthropic Claude Opus 4.5 (non-reasoning)	0.56	0.67	0.61
OpenAI gpt-oss-120B (reasoning)	0.45	0.67	0.54

Table 37. Binary evaluation metrics for detection of the "Rounding Errors" SQL antipattern using Few-Shot Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.83	1.00	0.91
Z.ai GLM-5 (reasoning)	1.00	1.00	1.00
Anthropic Claude Opus 4.5 (non-reasoning)	0.83	1.00	0.91
OpenAI gpt-oss-120B (reasoning)	0.77	1.00	0.87

Appendix 12 – Binary evaluation metrics for detection of individual SQL antipatterns using Chain-of-Thought Prompting

Table 38. Binary evaluation metrics for detection of the "Implicit Columns" SQL antipattern using Chain-of-Thought Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.92	0.96	0.94
Z.ai GLM-5 (reasoning)	0.94	0.93	0.94
Anthropic Claude Opus 4.5 (non-reasoning)	0.97	0.95	0.96
OpenAI gpt-oss-120B (reasoning)	0.88	0.89	0.89

Table 39. Binary evaluation metrics for detection of the "ID Required" SQL antipattern using Chain-of-Thought Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.91	0.94	0.93
Z.ai GLM-5 (reasoning)	0.92	0.95	0.94
Anthropic Claude Opus 4.5 (non-reasoning)	0.95	0.92	0.93
OpenAI gpt-oss-120B (reasoning)	0.86	0.96	0.91

Table 40. Binary evaluation metrics for detection of the "Keyless Entry" SQL antipattern using Chain-of-Thought Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.56	0.96	0.71
Z.ai GLM-5 (reasoning)	0.53	0.98	0.69
Anthropic Claude Opus 4.5 (non-reasoning)	0.72	0.98	0.83
OpenAI gpt-oss-120B (reasoning)	0.61	0.91	0.73

Table 41. Binary evaluation metrics for detection of the "Fear of the Unknown" SQL antipattern using Chain-of-Thought Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.49	0.50	0.49
Z.ai GLM-5 (reasoning)	0.60	0.25	0.35

Continues...

Table 41 – *Continues...*

Model	**Precision**	**Recall**	**F1-Score**
Anthropic Claude Opus 4.5 (non-reasoning)	0.88	0.39	0.54
OpenAI gpt-oss-120B (reasoning)	0.33	0.19	0.25

Table 42. Binary evaluation metrics for detection of the "31 Flavors" SQL antipattern using Chain-of-Thought Prompting.

Model	**Precision**	**Recall**	**F1-Score**
OpenAI GPT-5.2 (reasoning)	0.78	0.47	0.58
Z.ai GLM-5 (reasoning)	1.00	0.60	0.75
Anthropic Claude Opus 4.5 (non-reasoning)	1.00	0.60	0.75
OpenAI gpt-oss-120B (reasoning)	1.00	0.60	0.75

Table 43. Binary evaluation metrics for detection of the "Poor Man's Search Engine" SQL antipattern using Chain-of-Thought Prompting.

Model	**Precision**	**Recall**	**F1-Score**
OpenAI GPT-5.2 (reasoning)	0.62	0.87	0.72
Z.ai GLM-5 (reasoning)	0.64	0.67	0.65
Anthropic Claude Opus 4.5 (non-reasoning)	0.58	0.80	0.67
OpenAI gpt-oss-120B (reasoning)	0.24	0.67	0.36

Table 44. Binary evaluation metrics for detection of the "Rounding Errors" SQL antipattern using Chain-of-Thought Prompting.

Model	**Precision**	**Recall**	**F1-Score**
OpenAI GPT-5.2 (reasoning)	0.83	1.00	0.91
Z.ai GLM-5 (reasoning)	1.00	1.00	1.00
Anthropic Claude Opus 4.5 (non-reasoning)	1.00	1.00	1.00
OpenAI gpt-oss-120B (reasoning)	0.91	1.00	0.95

Appendix 13 – Binary evaluation metrics for detection of individual SQL antipatterns using Tree-of-Thought Prompting

Table 45. Binary evaluation metrics for detection of the "Implicit Columns" SQL antipattern using Tree-of-Thought Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.93	0.96	0.95
Z.ai GLM-5 (reasoning)	0.96	0.94	0.95
Anthropic Claude Opus 4.5 (non-reasoning)	0.96	0.94	0.95
OpenAI gpt-oss-120B (reasoning)	0.89	0.87	0.88

Table 46. Binary evaluation metrics for detection of the "ID Required" SQL antipattern using Tree-of-Thought Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.91	0.95	0.93
Z.ai GLM-5 (reasoning)	0.92	0.95	0.94
Anthropic Claude Opus 4.5 (non-reasoning)	0.96	0.92	0.94
OpenAI gpt-oss-120B (reasoning)	0.91	0.97	0.94

Table 47. Binary evaluation metrics for detection of the "Keyless Entry" SQL antipattern using Tree-of-Thought Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.65	0.95	0.77
Z.ai GLM-5 (reasoning)	0.57	0.98	0.72
Anthropic Claude Opus 4.5 (non-reasoning)	0.74	0.98	0.85
OpenAI gpt-oss-120B (reasoning)	0.64	0.88	0.74

Table 48. Binary evaluation metrics for detection of the "Fear of the Unknown" SQL antipattern using Tree-of-Thought Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.53	0.44	0.48
Z.ai GLM-5 (reasoning)	1.00	0.17	0.29

Continues...

Table 48 – *Continues...*

Model	Precision	Recall	F1-Score
Anthropic Claude Opus 4.5 (non-reasoning)	0.78	0.39	0.52
OpenAI gpt-oss-120B (reasoning)	0.14	0.11	0.13

Table 49. Binary evaluation metrics for detection of the "31 Flavors" SQL antipattern using Tree-of-Thought Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	1.00	0.60	0.75
Z.ai GLM-5 (reasoning)	0.89	0.53	0.67
Anthropic Claude Opus 4.5 (non-reasoning)	1.00	0.60	0.75
OpenAI gpt-oss-120B (reasoning)	0.88	0.47	0.61

Table 50. Binary evaluation metrics for detection of the "Poor Man's Search Engine" SQL antipattern using Tree-of-Thought Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	0.62	0.87	0.72
Z.ai GLM-5 (reasoning)	0.75	0.60	0.67
Anthropic Claude Opus 4.5 (non-reasoning)	0.55	0.80	0.65
OpenAI gpt-oss-120B (reasoning)	0.33	0.60	0.43

Table 51. Binary evaluation metrics for detection of the "Rounding Errors" SQL antipattern using Tree-of-Thought Prompting.

Model	Precision	Recall	F1-Score
OpenAI GPT-5.2 (reasoning)	1.00	1.00	1.00
Z.ai GLM-5 (reasoning)	1.00	1.00	1.00
Anthropic Claude Opus 4.5 (non-reasoning)	1.00	1.00	1.00
OpenAI gpt-oss-120B (reasoning)	1.00	1.00	1.00

Appendix 14 – Confusion matrices for individual SQL antipatterns with localisation

Table 52. Confusion matrix for detection of the "Implicit Columns" SQL antipattern using the developed tool.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	253	17	270
	Absent	4	N/A	4
	Total	257	17	274

Table 53. Confusion matrix for detection of the "Implicit Columns" SQL antipattern using the developed tool (corrected).

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	256	15	271
	Absent	1	N/A	1
	Total	257	15	272

Table 54. Confusion matrix for detection of the "ID Required" SQL antipattern using the developed tool.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	89	16	105
	Absent	13	N/A	13
	Total	102	16	118

Table 55. Confusion matrix for detection of the “ID Required” SQL antipattern using the developed tool (corrected).

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	101	7	108
	Absent	1	N/A	1
	Total	102	7	109

Table 56. Confusion matrix for detection of the “Keyless Entry” SQL antipattern using the developed tool.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	35	1	36
	Absent	30	N/A	30
	Total	65	1	66

Table 57. Confusion matrix for detection of the “Keyless Entry” SQL antipattern using the developed tool (corrected).

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	49	0	49
	Absent	16	N/A	16
	Total	65	0	65

Table 58. Confusion matrix for detection of the “Fear of the Unknown” SQL antipattern using the developed tool.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	19	17	36
	Absent	24	N/A	24
	Total	43	17	60

Table 59. Confusion matrix for detection of the “Fear of the Unknown” SQL antipattern using the developed tool (corrected).

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	42	17	59
	Absent	1	N/A	1
	Total	43	17	60

Table 60. Confusion matrix for detection of the “31 Flavors” SQL antipattern using the developed tool.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	38	1	39
	Absent	1	N/A	1
	Total	39	1	40

Table 61. Confusion matrix for detection of the “Poor Man’s Search Engine” SQL antipattern using the developed tool.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	13	8	21
	Absent	4	N/A	4
	Total	17	8	25

Table 62. Confusion matrix for detection of the “Rounding Errors” SQL antipattern using the developed tool.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	13	3	16
	Absent	0	N/A	0
	Total	13	3	16

Appendix 15 – Confusion matrices for individual SQL antipatterns without localisation

Table 63. Confusion matrix for detection of the "Implicit Columns" SQL antipattern using the developed tool without localisation.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	82	1	83
	Absent	4	N/A	4
	Total	86	1	87

Table 64. Confusion matrix for detection of the "Implicit Columns" SQL antipattern using the developed tool without localisation (corrected).

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	83	0	83
	Absent	3	N/A	3
	Total	86	0	86

Table 65. Confusion matrix for detection of the "ID Required" SQL antipattern using the developed tool without localisation.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	95	10	105
	Absent	5	N/A	5
	Total	100	10	110

Table 66. Confusion matrix for detection of the "ID Required" SQL antipattern using the developed tool without localisation (corrected).

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	100	5	105
	Absent	0	N/A	0
	Total	100	5	105

Table 67. Confusion matrix for detection of the "Keyless Entry" SQL antipattern using the developed tool without localisation.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	18	0	18
	Absent	26	N/A	26
	Total	44	0	44

Table 68. Confusion matrix for detection of the "Keyless Entry" SQL antipattern using the developed tool without localisation (corrected).

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	34	0	34
	Absent	10	N/A	10
	Total	44	0	44

Table 69. Confusion matrix for detection of the "Fear of the Unknown" SQL antipattern using the developed tool without localisation.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	15	2	17
	Absent	28	N/A	28
	Total	43	2	45

Table 70. Confusion matrix for detection of the “Fear of the Unknown” SQL antipattern using the developed tool without localisation (corrected).

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	32	2	34
	Absent	11	N/A	11
	Total	43	2	45

Table 71. Confusion matrix for detection of the “31 Flavors” SQL antipattern using the developed tool without localisation.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	15	0	15
	Absent	0	N/A	0
	Total	15	0	15

Table 72. Confusion matrix for detection of the “Poor Man’s Search Engine” SQL antipattern using the developed tool without localisation.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	10	1	11
	Absent	0	N/A	0
	Total	10	1	11

Table 73. Confusion matrix for detection of the “Rounding Errors” SQL antipattern using the developed tool without localisation.

		Predicted Presence		
		Present	**Absent**	Total
Actual Presence	**Present**	9	6	15
	Absent	0	N/A	0
	Total	9	6	15

Appendix 16 – Project-level Jaccard Index

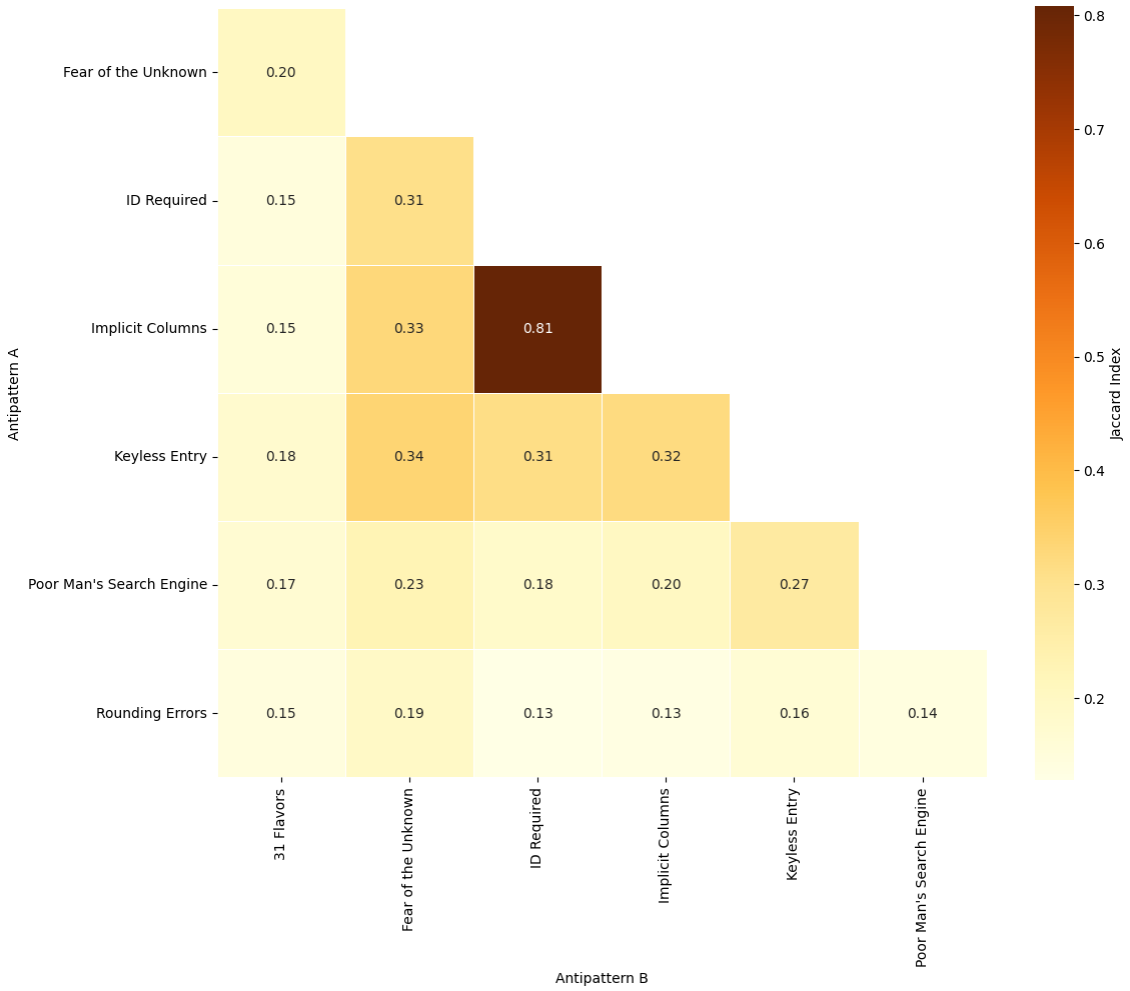

Figure 43. Project-level Jaccard Index.

Appendix 17 – File-level Jaccard Index

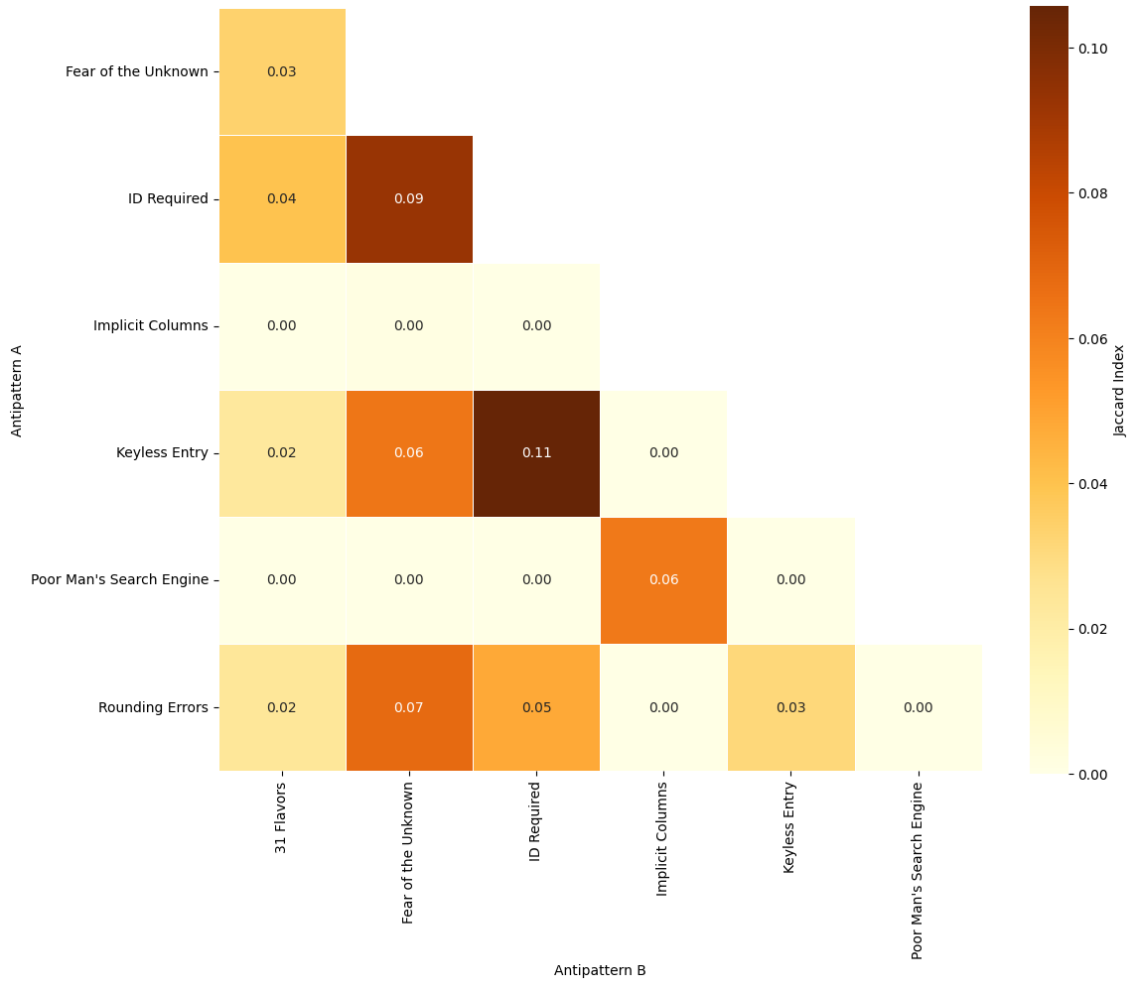

Figure 44. File-level Jaccard Index.

Appendix 18 – Project-level Conditional Probability

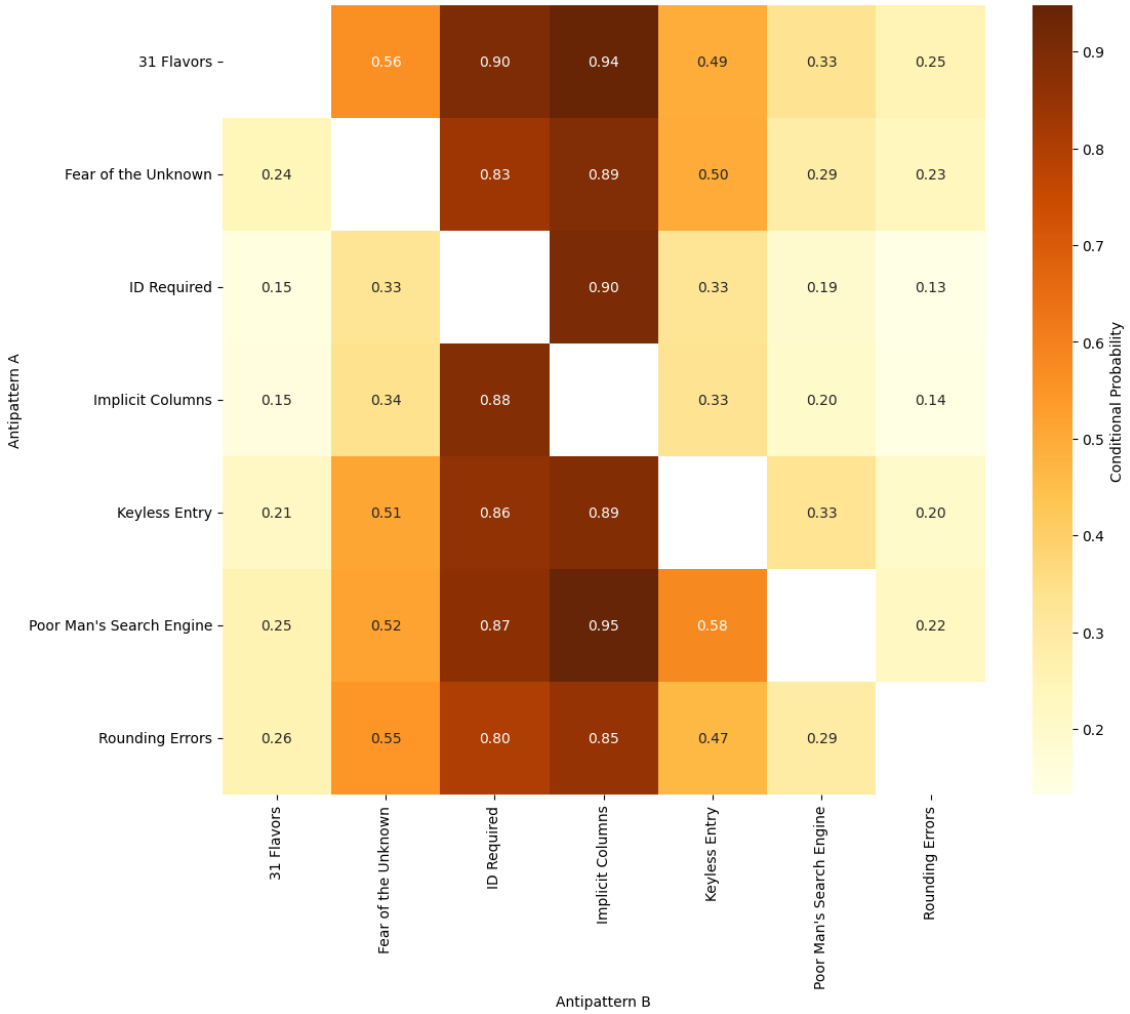

Figure 45. Project-level Conditional Probability.

Appendix 19 – File-level Conditional Probability

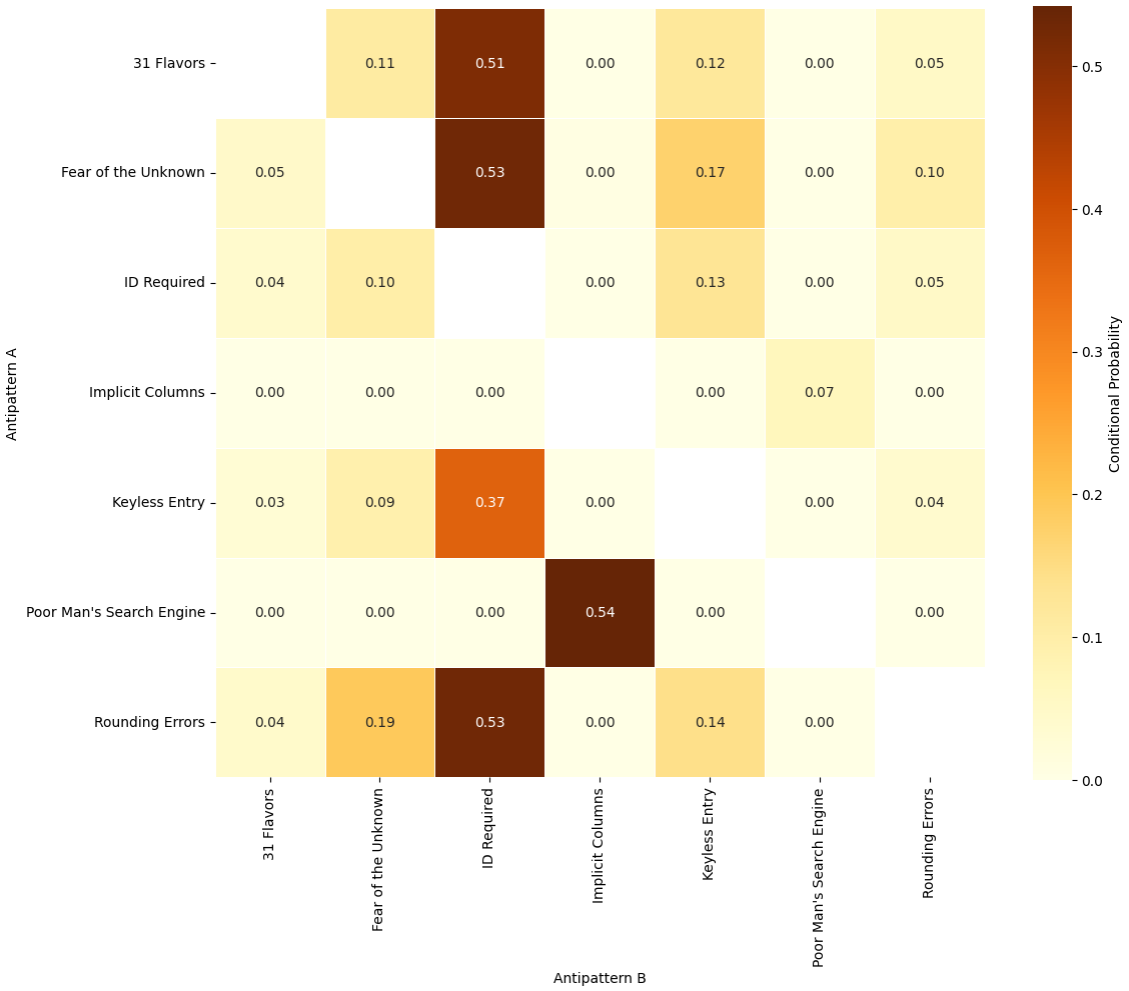

Figure 46. File-level Conditional Probability.

Appendix 20 – Project-level Spearman Correlation

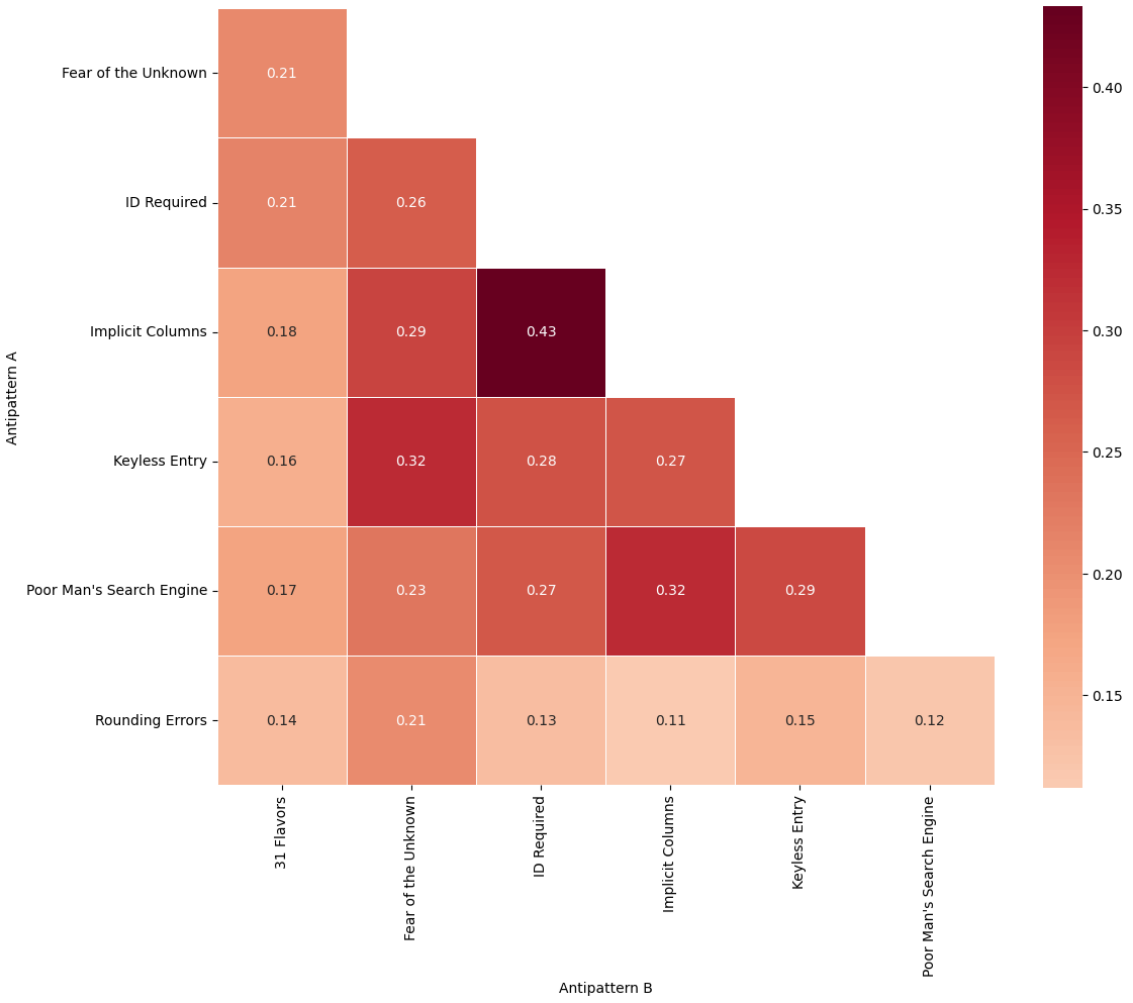

Figure 47. Project-level Spearman Correlation.

Appendix 21 – File-level Spearman Correlation

Figure 48. File-level Spearman Correlation.