Ca' Foscari
University
of Venice

Single Cycle Degree programme

in Data Analytics
for Business and
Society

Final Thesis

Generative Business Intelligence in Telecommunications

A Local LLM and RAG-Based Framework for Secure Text-to-SQL Analytics

Supervisor
Ch.Prof. Luca Cosmo

Graduand
Alsadig Elfatih Osman Gambour
Matriculation Number 908557

Academic Year
2025 / 2026

Abstract

Traditional business intelligence in telecommunications was built for descriptive, retrospective reporting, relying on batch ETL pipelines, static thresholds, and predefined dashboards. As network complexity, data volume, and customer expectations have grown, these systems have struggled to deliver timely, contextual insight, producing alert fatigue, slow problem discovery, vendor silos, and reactive churn management. Generative Business Intelligence (GenBI), enabled by large language models, retrieval-augmented generation (RAG), and agentic workflows, offers a shift from passive reporting toward active, natural-language analytics. This thesis investigates how Generative BI can be designed and deployed for a telecommunications operator under stringent privacy and on-premise constraints. It develops a seven-phase framework spanning semantic-layer construction, analyst-question example design, time-intelligence handling, model selection, deployment, evaluation, and data governance, and applies it to the text-to-SQL task over a synthetic telecom schema. Two candidate models — a general-purpose cloud API (GPT-3.5) and an open-source, locally deployable specialist (Defog SQLCoder-7B) are compared on execution success rate, result correctness, runtime, hallucination, and privacy. In this proof-of-concept evaluation, the locally deployed specialist matched or exceeded the cloud model on execution success rate and result correctness, produced no schema-ungrounded queries, ran at comparable speed, and uniquely satisfied the operator's data-locality obligations under the GDPR. The thesis contributes a reproducible methodology for telecom Generative BI deployment that respects data-controller obligations while opening natural-language analytics to non-technical stakeholders.
Keywords: Generative BI, telecommunications, text-to-SQL, retrieval-augmented generation, on-premise LLM, data governance, GDPR.

Table of Contents

Chapter 1 Introduction

1.1 Background and Definitions

The telecommunications industry has become one of the core systems that keeps modern life connected. Telecom operators were once seen mainly as utility providers — companies that built towers, laid cables, and kept calls moving. Today, they are also large data-driven organizations. Every mobile connection, data session, text message, service subscription, network alarm, and customer interaction produces information. To understand the problem addressed in this thesis, it is useful to begin with a simple question: how can such a large organization turn all of that stored data into timely and trustworthy business insight?

Before data can support decisions, it must be stored and organized. At the simplest level, a database is a structured digital filing system. In large enterprises, this usually means relational databases, where information is arranged in tables of rows and columns. However, numbers sitting in tables are not useful on their own. Business Intelligence (BI) is the process that turns fragmented operational data into reports, trends, dashboards, and decisions that managers can act on. For many years, this process has depended on ETL pipelines. These pipelines extract data from operational systems, clean and reshape it, and load it into a central warehouse for analysis (Panintelligence, 2025).

In telecommunications, this standard BI model becomes difficult very quickly because the data is both massive and fragmented. Business Support Systems (BSS) hold the commercial side of the operator, including billing, subscriptions, marketing campaigns, and customer-service records. Operations Support Systems (OSS) hold the technical side, including network performance, alarms, latency, configuration, and service availability. Between these two worlds sits the Call Detail Record (CDR), which is created whenever a subscriber makes a call, sends a message, or uses mobile data. A CDR records details such as time, duration, origin, destination, and usage. A mid-sized operator can generate billions of these records each day. This scale explains why telecom analytics is not simply a reporting challenge; it is a data-access and interpretation challenge.

1.2 Problem Statement

The central problem in this thesis is that telecom operators are data-rich but often insight-poor. The data needed to answer commercial and technical questions already exists inside the organization but reaching that answer is usually slow. Business managers and analysts may know exactly what they want to ask, but many of them cannot write SQL. As a result, they depend on specialized data teams to translate each business question into a database query. This turns even routine requests into queues and delays. Generative BI promises to reduce this bottleneck by allowing users to ask questions in natural language while the system generates the SQL needed to answer them.

In a telecom environment, however, this promise faces two serious obstacles. The first is correctness. A general-purpose language model does not automatically understand an operator's own metric definitions, database schema, or time logic. If it is asked about ARPU, churn, revenue movement, or year-over-year performance, it may produce SQL that looks convincing but applies the wrong logic. In telecom analytics, this is dangerous because metrics are often governed by strict internal definitions. A polished but incorrect answer can mislead decision-makers more than no answer at all. The second obstacle is privacy. Many powerful models are accessed through public cloud APIs, which can

require query content or sensitive subscriber-related information to leave the operators-controlled environment. For telecom operators working under GDPR-style obligations, that is not just a technical concern; it is a governance risk.
This thesis therefore asks how the Generative BI capability can be built in a way that is both accurate enough for telecom-specific analytics and private enough to run within the operator's own environment. The aim is not only to generate SQL, but to generate SQL that reflects real telecom definitions, follows the correct schema, and protects sensitive data.

1.3 Generative BI

Generative Business Intelligence changes the way users interact with data (Panintelligence, 2025). In traditional BI, users mainly consume dashboards that were designed in advance. Those dashboards are useful when the question is predictable, but they are less helpful when a manager wants to explore a new issue quickly. Generative BI makes the interaction more conversational. Instead of waiting for a new report, a user can ask a question directly and continue with follow-up questions as the analysis develops.
The technology behind this shift is the large language model (LLM), an AI system trained on large volumes of text and code. Because these models learn patterns in language, they can interpret a user's question and produce a fluent response. In analytics, their value is not only that they can write text, but that they can translate a business question into a structured database query.
In a Generative BI workflow, this capability supports Natural Language Querying (NLQ). A non-technical user can ask a question in plain English, and the system can translate that question into SQL, run it against the database, and return a plain-language explanation of the result. For telecom operators, this could reduce the distance between business users and the data they need every day.

1.4 Retrieval-Augmented Generation (RAG)

Large language models are powerful, but they are not automatically reliable. When they do not have enough task-specific knowledge, they can produce answers that sound confident while being unsupported or wrong. This problem is known as hallucination (Ji et al., 2023). In a telecom BI setting, hallucination may appear as a non-existent table, a wrong column, an incorrect join, or a metric formula that does not match the operator's definition.
Retrieval-Augmented Generation (RAG) is one way to reduce this risk. It works like giving the model an approved reference pack before it answers. When a user asks a question, the system first retrieves relevant documents, definitions, schema descriptions, or examples from a trusted knowledge base. The model then uses that retrieved context to generate its answer or SQL query (Techplayon, 2025). In this process, the knowledge base is often stored in a vector database, where passages are represented as numerical embeddings that capture meaning. This allows the system to retrieve relevant material even when the user's words do not exactly match the wording in the documents. For example, a question about "customers leaving" can still retrieve a definition of churn.

1.5 Generative BI Significance to the Telecommunications Sector

The importance of Generative BI in telecommunications comes from real operational pressure, not from technology fashion. Modern telecom networks are complex, virtualised, and fast-moving. Thousands of network elements produce telemetry continuously, while business systems generate billing, subscription, usage, and customer-interaction data at scale. Traditional dashboards cannot anticipate every

possible question or network scenario. When dashboards and static rules are stretched too far, engineers and analysts may face alert overload, delayed problem discovery, and missed links between technical incidents and commercial impact (Anodot, 2025).

1.6 Research Objectives and Scope

The primary objective of this research is to develop, deploy, and critically evaluate a RAG-based Generative BI framework designed for the telecommunications sector.

To achieve this objective, the thesis focuses on three sub-objectives. The first is architectural: to design a practical framework for building a telecom-specific semantic layer that helps the model understand telecom metrics, schema relationships, and business terminology.

The second is evaluative: to move beyond simple pass-or-fail accuracy and examine why models succeed or fail. This includes comparing a general-purpose cloud model with a locally deployable, text-to-SQL specialized model and studying the types of errors each model produces.

The third is performance-focused: to measure execution success, result correctness, end-to-end runtime, hallucination behaviour, and privacy compliance under a controlled proof-of-concept setup.

The scope of the study is the analytical layer of telecommunications, specifically the ability to translate natural-language questions into SQL queries. Because real telecom environments contain very large and sensitive datasets, the evaluation is conducted as a proof-of-concept using a representative schema and a labelled set of fifty business queries. These queries are divided into three difficulty levels: twenty simple, fifteen moderate, and fifteen complex queries. This design is intentionally balanced. It is large enough to report meaningful internal-validation proportions, while still small enough to allow detailed qualitative inspection of how each model generates SQL and where it fails.

Chapter 2 Literature Review

2.1 Introduction

This chapter reviews the work most relevant to secure Generative BI in telecommunications. It starts with traditional telecom BI and its dependence on ETL pipelines, dashboards, and retrospective reporting. It then explains how large language models created a path toward natural-language analytics, before turning to Retrieval-Augmented Generation as a way to reduce hallucination and ground answers in trusted sources. The chapter also discusses the privacy limits of cloud-based LLM APIs and why local models are more suitable for telecom operators. Finally, it reviews examples of AI adoption in the telecom industry and identifies the research gap addressed by this thesis.

2.2 Traditional Business Intelligence in Telecommunications

Telecommunications operators were once viewed mainly as infrastructure companies. Their early data work focused on back-office reporting, such as finance summaries, subscriber counts, and capacity planning. As mobile networks expanded, operators became major data producers, and Business Intelligence became essential for turning that data into operational and commercial insight.

The backbone of this traditional BI model is the ETL pipeline. Data is extracted from billing systems, customer-care platforms, and network elements, then cleaned, standardized, and loaded into a data warehouse for reporting. This approach supports descriptive analytics: it explains what has already happened. A revenue manager may

review churn at month-end, or a network engineer may inspect outage reports after an incident. The process is useful, but it is naturally reactive.

This model struggles in modern telecom environments. Networks are now virtualized and software-defined, producing constant streams of telemetry that static dashboards cannot fully anticipate. At the same time, BSS and OSS data often remain separated, so it can be difficult to connect a technical issue, such as dropped calls in a region, with its business impact, such as churn or revenue loss. The result is a familiar bottleneck: business users ask questions, data teams translate them into SQL, and answers arrive later than needed. Generative BI aims to reduce that gap by allowing authorized users to ask questions directly in natural language.

2.3 From Large Language Models to Generative Business Intelligence

The limits of dashboard-based BI led naturally to a new question: instead of building every possible chart in advance, could users simply ask the database what they need to know? This is the idea behind Natural Language Querying. Earlier systems relied on rules or trained semantic parsers, and benchmarks such as WikiSQL and Spider helped measure progress in text-to-SQL. However, these systems still struggled with the complexity and domain-specific language found in telecom databases.

Large language models changed the situation. Models such as GPT-3.5, GPT-4, and Llama showed that a general-purpose model could often generate SQL from a plain-English question when given the database schema. This is the basis of Generative BI: the model interprets the question, writes the SQL, and can also explain the result in ordinary language.

For telecom operators, this creates a more direct route from question to insight. A manager could ask why ARPU dropped in a region, or which customer segment generated the most tickets, without waiting for a custom dashboard or analyst ticket. In this sense, Generative BI does not replace dashboards; it adds a more flexible access layer on top of the data warehouse.

2.4 Retrieval-Augmented Generation and the Hallucination Problem

The main risk in LLM-driven BI is hallucination. When a model does not know the correct answer, it may still produce a confident-looking response. In telecom analytics, this could mean using the wrong KPI definition, inventing a column, or applying the wrong time logic. Because metrics such as ARPU, churn, and revenue are defined precisely, a confident but wrong answer can mislead decision-makers.

Retrieval-Augmented Generation is widely seen as a practical way to reduce this risk. Instead of asking the model to rely only on memory, the system retrieves relevant material from trusted sources such as KPI definitions, schema descriptions, vendor manuals, billing rules, or analyst examples. The retrieved content is then placed into the prompt so that the model can answer from verified context rather than guessing.

Vector databases make this retrieval possible. Instead of matching only exact words, they compare the meaning of a user's question with the meaning of stored passages. This matters in telecom because users may say "customers leaving" while the official document says "churn." Vector search can still connect the two. Because the retrieval layer can also run locally, the operator can keep documents and embeddings inside its own environment.

RAG helps in three ways. It grounds the answer in trusted text, keeps knowledge current when documents are updated, and allows smaller local models to perform better on domain-specific tasks. For telecom Generative BI, this is not an optional enhancement. It is one of the main safeguards that makes natural-language analytics reliable enough for business use.

A clear example is ARPU. It may sound like a simple average, but in practice it depends on the operator's own definition of revenue and active subscribers. Without the correct definition, a model may generate a query that looks reasonable but is wrong. By retrieving the exact KPI definition at query time, RAG helps the model apply the operator's real business logic.

2.4.1 How RAG Works

RAG works in three stages. The first is indexing: internal documents — vendor PDFs, KPI definitions, network manuals, billing rules — are broken into short passages, and each passage is turned into a numerical vector, an embedding, that is stored in a vector database.

Retrieval comes next. The user's question is mapped into that same vector space, and the system returns the few passages whose meaning sits closest to it. The final stage is generation: those passages are placed into the model's prompt as context, and the model is told to answer from that evidence alone (Lewis et al., 2020; Techplayon, 2025).

2.4.2 Advantages of RAG

- Reduces hallucinations: the AI must read the document before answering, so it cannot make things up (Lewis et al., 2020).
- Keeps knowledge fresh: update the document and the AI is up to date right away, with no retraining.
- Preserves data privacy: customer data stays inside the company and the AI only looks at it when answering a question (IBM, 2025).
- Auditable: every answer can be traced back to its source document, which is what GDPR regulators need to see.
- Cost-efficient: a changed rule means updating one document, not paying to retrain the model (IBM, 2025).

2.4.3 How RAG Helps LLM Models

RAG extends what an LLM can use at the moment of answering. A standard model is limited by its training data and does not know an operator's current schema, KPI catalogue, or internal documents. RAG supplies that information at query time. This reduces fabrication, improves relevance, and allows a smaller locally hosted model to behave more like a domain-aware assistant.

2.5 The Privacy Problem with Cloud LLM APIs

Many leading LLMs are accessed through public cloud APIs. This is convenient, but it creates a significant issue for telecom operators. Subscriber identifiers, billing data, usage records, call detail records, and location-related information are sensitive personal data. Sending this information to an external inference endpoint can conflict with the operator's duties under privacy regulations such as the GDPR, especially the requirement to protect personal data against unauthorized or unlawful processing.

A more suitable deployment model is to run an open-weights model inside the operator's own environment. Models such as Llama, Mistral, Qwen, and SQLCoder can be hosted on-premises or in a private cloud and combined with a local RAG layer. This keeps data under the operator's control. The trade-off is that smaller local models may have less raw capability than the largest cloud models, but RAG reduces that gap by providing the exact context needed for each query.

This is the architecture this thesis evaluates: a local, open-source LLM, paired with a telecom-specific Retrieval-Augmented Generation layer, deployed behind the operator's own firewall.

Two practical components support this local architecture. The first is the semantic layer, which maps telecom business terms such as ARPU, churn, and active subscriber to the

correct database tables and columns. The second is RAG over unstructured documents, such as vendor manuals and internal policy files. Together, they allow the local model to reason with telecom-specific knowledge without exposing subscriber data.

2.6 Industry Adoption: Real-World Examples

While much of the academic literature focuses on theory, several major operators have already begun integrating AI into their BI and analytics functions, providing concrete evidence that the shift away from static dashboards is underway.

Vodafone has deployed an internal generative AI assistant called "SuperAgent," built in partnership with Microsoft and Accenture, to help its customer-service and business teams query operational data and resolve customer issues more quickly. The platform combines LLMs with the company's own knowledge bases through retrieval techniques, allowing agents to ask questions in natural language instead of navigating multiple dashboards (Vodafone, 2024).

AT&T has built an internal platform called "Ask AT&T," a generative AI tool used across software development, network operations, and business analytics. The system is hosted on the operator's private Azure environment to keep customer data inside the company's security perimeter, which addresses the same GDPR-style privacy constraints discussed earlier in this chapter (AT&T, 2023).

Deutsche Telekom has launched "askT," an internal LLM-based assistant available to its global workforce and is collaborating with Google Cloud and SAP on AI-driven customer-experience and network-analytics products. The operator has publicly stated that AI is now a core component of its T-Systems data and analytics offering (Deutsche Telekom, 2024).

SK Telecom in South Korea has gone further by building its own telecom-specific large language model, "A.X," trained on Korean-language and telecom-domain data and used to power BI assistants, contact-center automation, and network-operations copilots. SK Telecom has also joined the Global Telco AI Alliance with Deutsche Telekom, e&, Singtel, and SoftBank to co-develop telecom-tuned LLMs (SK Telecom, 2024).

These examples show that telecom operators are already moving beyond static dashboards. The common direction is clear: LLMs and retrieval-based systems are being integrated into analytics and operations, usually within controlled environments. This supports the local-RAG approach developed in this thesis.

2.7 Synthesis and the Research Gap

The literature supports the value of Generative BI, RAG, and local models for telecom analytics, but it leaves an important gap. Many studies rely on generic benchmarks that do not reflect telecom metrics, telecom schemas, or operator privacy constraints. They also tend to report accuracy without explaining why models fail. This thesis addresses that gap by evaluating a telecom-specific text-to-SQL framework on a structured fifty-query set and by examining model errors qualitatively. The next chapter builds the technical foundation needed to understand that evaluation.

Chapter 3 Technical Foundations: Large Language Models and Text-to-SQL

3.1 Introduction and Roadmap

This chapter explains the technical foundations behind the system evaluated in the thesis. It is written for readers who understand data and telecommunications but may not specialize in machine learning. The chapter explains what language models are, how transformers work, how models are trained, and why text-to-SQL is difficult. It then compares GPT-3.5 and SQLCoder, the two models evaluated later in the study.

The chapter moves from basic concepts to the specific comparison used in this thesis. It first defines language models and transformers, then describes training and adaptation, then frames text-to-SQL as a technical task. Finally, it explains why a general-purpose cloud model and a specialized local model are expected to behave differently in a telecom BI setting.

3.2 What a Language Model Is

A language model predicts the next piece of text from the text that came before it. Modern models use these tokens, which are often parts of words rather than full words. A simple analogy is predictive text on a phone: after seeing “the customer wants to check their data,” the model treats “balance” or “usage” as more likely than an unrelated word. The same next-token prediction capability allows a model to answer questions, write explanations, and generate SQL.

Language models are not new. For decades the dominant approach was statistical: so-

called n-gram models estimated the probability of the next word by counting how often short sequences of words had occurred together in a body of text. These models were useful but shallow. They had no memory beyond the last few words, they could not generalize to phrasings they had never seen, and they grew unwieldy as the sequences they tracked got longer. The arrival of neural language models, which represent words as dense numerical vectors and learn from data how those representations combine, removed some of these limits, and recurrent neural networks could in principle carry information across longer spans of text. In practice, however, recurrent models struggled to learn long-range dependencies (Bengio et al., 1994) and were slow to train because they processed text strictly in sequence.

The biggest breakthrough was the transformer architecture, introduced in 2017. As transformer models became larger and were trained on more data, they began to perform tasks beyond simple text completion, including instruction following and code generation. This is why modern LLMs can attempt text-to-SQL: they have seen enough language and code to learn patterns that connect questions with structured queries.

3.3 The Transformer Architecture

The transformer is important because it handles context differently from older models. Instead of reading text strictly one word at a time, it processes a sequence of tokens together. Each token is converted into an embedding, and positional information is added so the model understands order. This allows the model to consider both the meaning of each token and its place in the sentence.

The key mechanism is self-attention. It lets the model decide which parts of the input matter most for each token. For example, in the question “revenue per region year over year,” the model must connect revenue, region, and year-over-year comparison. Self-attention makes this kind of long-range connection possible, which is especially important when generating SQL.

Transformers use many layers and attention heads. Each layer refines the model’s representation of the input, and different attention heads can focus on different relationships. The model’s learned values are called parameters. A model with billions of parameters has more capacity, but it also requires more computation to run.

The models in this study are decoder-only transformers. They generate output one token at a time. A SQL query is produced the same way: the model predicts the next token, adds it to the query, and continues until the query is complete. This explains why early mistakes matter. If the model chooses the wrong table or misses a window function, the rest of the query may still look coherent but return the wrong result.

Generation settings also matter. A higher temperature introduces randomness, which can be useful for creative writing but is risky for SQL. This study uses low or deterministic settings so the same question produces the same query as consistently as possible. That makes evaluation fairer and more reproducible.

3.3.1 Tokenization, Embeddings, and the Context Window

Before any of the attention machinery described above can operate, the input text must be converted into a form the model can process. A language model does not read words or characters directly but instead it splits text into tokens, the sub-word units that form its working vocabulary. Most modern models use a sub-word scheme such as byte-pair encoding (Sennrich et al., 2016), which represents common words as single tokens while breaking rarer strings into smaller fragments. This matters for the present task, because domain terms and schema identifiers — an abbreviation such as ARPU, or a column name such as subscriber_id — are typically fragmented into several tokens rather than recognized as atomic units. The number of tokens a passage occupies, rather than the

number of words, is what determines both the computational cost of processing it and how much of the model's finite input budget it consumes.
Each token is then mapped to a high-dimensional vector known as an embedding. Embeddings position tokens in a continuous space in which proximity reflects semantic and functional similarity, so that terms used in similar contexts come to occupy nearby regions (Mikolov et al., 2013). It is on these vectors, not on the surface text, that the self-attention mechanism operates, computing relationships between tokens as geometric relationships between their embeddings. The same principle underpins the retrieval stage of the framework examined later in this chapter: passages of the operator's semantic layer are themselves embedded as vectors, and relevance is judged by vector proximity rather than by exact keyword matching.
Finally, a model can attend to only a bounded number of tokens at any one time, a limit known as the context window. Everything the model needs for a single request must fit inside it: the system instruction, any retrieved schema and semantic-layer content, the user's question, and the SQL the model is to generate. This is a hard architectural constraint with a direct bearing on retrieval-augmented generation, because the grounding context inserted into the prompt competes for the same budget as the question and the answer. Retrieval must therefore be selective, supplying the most relevant schema elements rather than the entire database definition; a larger context window relaxes this pressure but increases both the computational cost and the latency of each query.

3.4 How Large Language Models Are Trained

Large language models are trained in stages. The first stage is pre-training, where the model learns from huge collections of text and code by predicting the next token. This gives the model broad language ability and some knowledge of programming and SQL syntax. However, a pre-trained model is not automatically a helpful assistant; it has learned patterns, not a specific user-facing task.
Instruction tuning teaches the model to respond to requests more directly, while reinforcement learning from human feedback improves the usefulness and style of the responses. These steps help create systems such as GPT-3.5. Even so, the model does not automatically know a telecom operator’s schema, internal KPI definitions, or governance rules.
There are two main ways to adapt a model to a domain. The first is fine-tuning, where the model’s parameters are trained further on task-specific examples. SQLCoder follows this path by being fine-tuned on many natural-language-to-SQL examples. The second is prompting or retrieval-based grounding, where the model’s parameters stay unchanged but the prompt provides schema information, definitions, and examples at query time. This thesis relies on the second method through RAG and schema-aware prompting. These two methods can work together. Fine-tuning teaches a model how SQL generally works; retrieval tells it what this specific database and telecom environment look like. The framework evaluated in this thesis combines a specialized local text-to-SQL model with retrieved schema and semantic context.

3.5 Text-to-SQL as a Technical Task

Text-to-SQL is the task of translating a natural-language question into an executable SQL query. It sounds simple, but it is difficult because natural language is flexible while SQL is strict. A user can ask an imprecise question, but the database requires exact table names, column names, joins, filters, and calculations. A query that is almost correct can still return the wrong answer.

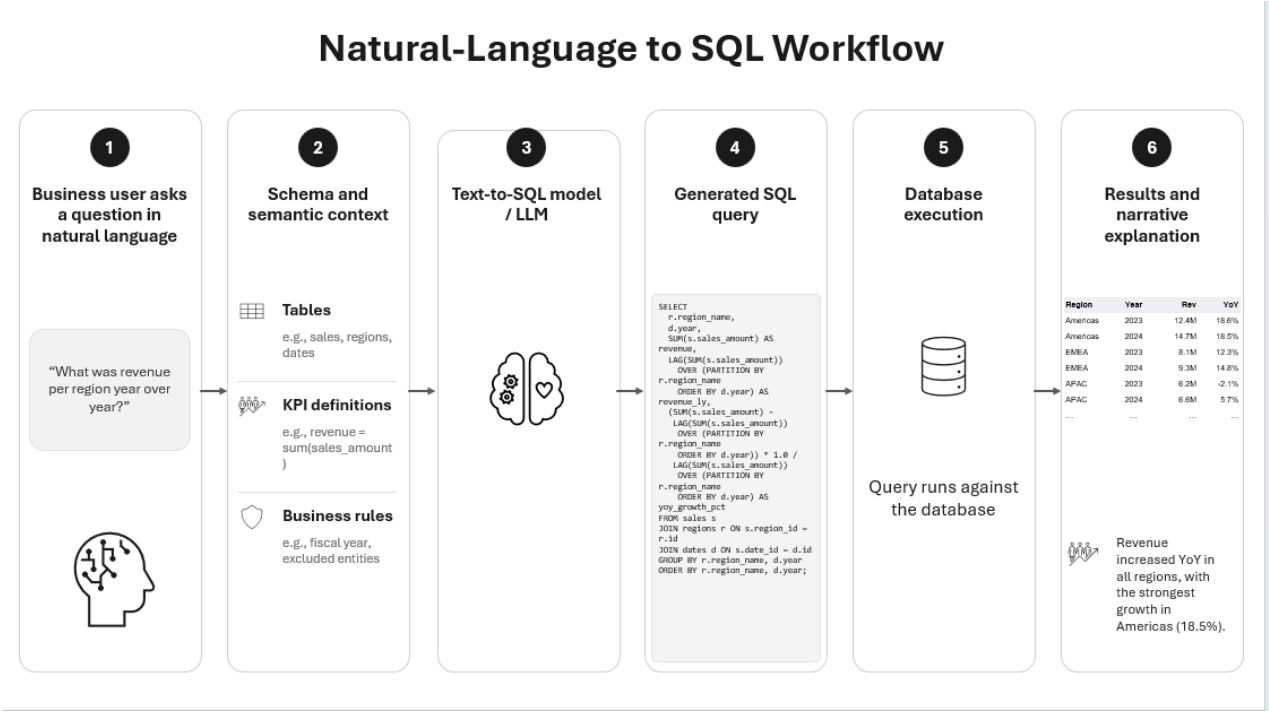

Figure 3.1 — The natural-language-to-SQL workflow: from a business question through schema and semantic context to a generated SQL query, database execution, and a narrated result.

The first challenge is schema linking. The model must connect words in the user's question to the right tables and columns. For example, "revenue per region" may require data from revenue, subscriber, and region tables. The wording used by analysts rarely matches the database schema exactly, so the model needs clear schema grounding.
The second challenge is composition. Real telecom questions often combine joins, filters, aggregations, grouping, and time comparisons. A year-over-year revenue query by region, for example, needs the comparison to be calculated separately for each region. Missing that partition can make the query run successfully while returning the wrong answer.
A third challenge is SQL dialect. A query that works in one database may fail in another. This study evaluates generated SQL against the selected reference database, so correctness depends not only on logical meaning but also on whether the generated query is valid for that environment.

3.5.1 The Retrieval-Augmented Generation Pipeline

The conceptual case for retrieval-augmented generation was set out in Section 2.4; here the mechanism is described as the concrete pipeline that the framework implements, illustrated end to end in Figure 3.2. RAG operates in two stages. The first is an offline indexing stage, performed once before the system is used. The operator's semantic layer — the schema descriptions, metric definitions, and worked question-to-SQL examples assembled in Chapter 4 — is divided into discrete passages, or chunks, each small enough to be retrieved and reasoned over independently. Every chunk is converted into an embedding using the vector representation described in Section 3.3.1 and stored in a vector index (Lewis et al., 2020). Chunking granularity is a design decision in itself: passages that are too large dilute relevance and waste context budget, while passages that are too small fragment the schema relationships the model needs to construct as correct join.
The second stage runs for every query. The analyst's question is embedded with the same model used to build the index and compared against the stored vectors by similarity, a technique known as dense retrieval (Karpukhin et al., 2020); the index

returns the top-ranked passages whose embedding lie closest to the question. Because exhaustive comparison would not scale to a large semantic layer, this search is performed with approximate nearest-neighbor methods that trade a negligible loss of recall for a large gain in speed (Johnson et al., 2019). Retrieval is deliberately selective, returning only the handful of schema elements most relevant to the question rather than the entire database definition, both to respect the context window and to avoid distracting the model with irrelevant tables.

Finally, the retrieved passages are assembled, together with the system instruction and the user's question, into a single prompt that fits within the context window discussed in Section 3.3.1. The language model then generates SQL conditioned on this grounded context. The decisive consequence is that the table names, column names, and metric definitions the model relies on are supplied as retrieved fact rather than recalled from parametric memory, which is precisely the property that constrains the hallucination problem: a model that has shown the schema has far less occasion to invent one.

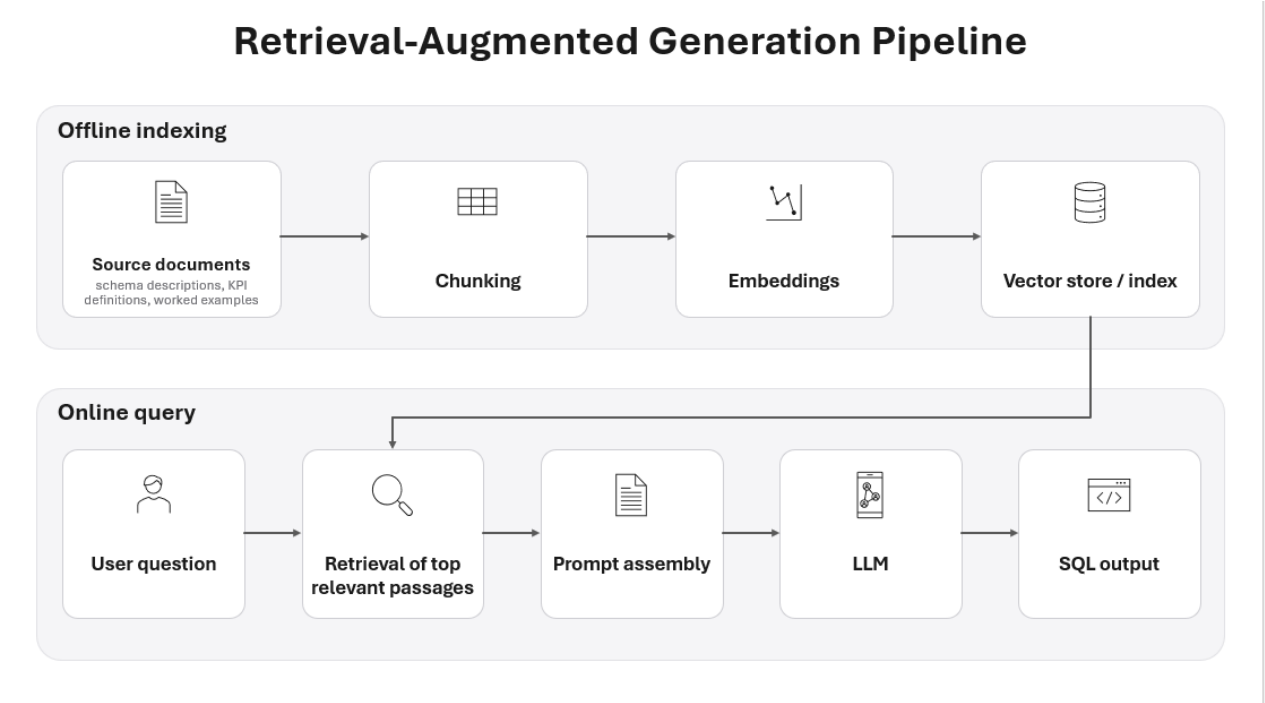

Figure 3.2 — The retrieval-augmented generation pipeline: an offline indexing stage (chunking, embedding, and vector storage of the semantic layer) and an online query stage (retrieval, prompt assembly, and SQL generation).

3.6 General-Purpose and Specialized Models: GPT-3.5 and SQLCoder

The two models compared in this study sit at opposite ends of a spectrum, and the choice to evaluate them head-to-head is deliberate. GPT-3.5, accessed through OpenAI's commercial API, is a general-purpose model. It descends from the GPT-3 family, with its largest variant holding on the order of one hundred and seventy-five billion parameters, and it reaches the public as a product that has been instruction-tuned and aligned through reinforcement learning from human feedback. Its strengths are breadth and fluency: it handles an enormous range of tasks competently, text-to-SQL among them, without ever having been built specifically for any one of them. For the present use case it carries two structural disadvantages, however. It was never specialized for SQL generation, so it brings only the general competence absorbed during pre-training, and, more consequentially, it runs only on OpenAI's infrastructure, which means that to use it the operator must send query content out of its own network.

Defog's SQLCoder represents the opposite design philosophy. It is an open-weights

model built specifically for the text-to-SQL task, available in a family of sizes, and each member is produced by fine-tuning an open base model, the original fifteen-billion-parameter version on StarCoder and the seven-billion-parameter versions on code-oriented bases such as CodeLlama and Mistral, on more than twenty thousand human-curated natural-language-to-SQL examples spanning ten distinct schemas. The variant used in this study, referred to here as SQLCoder-7B, is a roughly seven-billion-parameter model, an order of magnitude smaller than GPT-3.5, yet on Defog's own evaluation framework the SQLCoder models match or exceed GPT-3.5 on SQL generation, a direct illustration of how specialization can outweigh raw scale. Because its weights are openly licensed under CC-BY-SA-4.0, the model can be downloaded and run entirely on the operator's own hardware, and it is designed expressly for read-only analytical workloads of the kind a business intelligence system serves.

Although the two models share the same underlying transformer machinery, they differ along three axes that decide the comparison for a telecom operator. The first is specialization: a general-purpose model against one whose parameters have been tuned for SQL. The second is openness and deployment: a proprietary model reachable only through a cloud API against an open model that can be hosted locally. The third is the route by which domain knowledge enters the model: GPT-3.5 must be told about the schema in the prompt and otherwise relies on its frozen, generic pre-training, whereas SQLCoder combines schema grounding with parameters already shaped by extensive SQL training. Table 3.1 sets these and related properties side by side.

Dimension	GPT-3.5 (OpenAI)	SQLCoder (Defog, 7B)
Architecture	Decoder-only transformer	Decoder-only transformer
Specialization	General-purpose language model	Fine-tuned specifically for text-to-SQL
Base lineage	GPT-3 model family	Fine-tuned from an open base
Parameters	Approx. 175 billion (largest variant)	Approx. 7 billion
Adaptation	Instruction tuning and RLHF	Supervised fine-tuning on 20,000+ curated NL-to-SQL pairs over 10 schemas
Weights and license	Proprietary, closed	Open weights, CC-BY-SA-4.0
Deployment	Cloud API only (off-premises)	Fully on-premises
Domain knowledge	Schema supplied in the prompt at inference	Schema grounding plus parameters tuned for SQL
Data residency	Query content leaves the operator's network	Data remains on operator infrastructure

Table 3.1 — Architectural, training, and deployment comparison of the two candidate models.

3.6.1 The Wider Field of Candidate Models

Although the head-to-head evaluation in this study is confined to the two models in Table 3.1, they were selected from a wider field of candidates. Three criteria governed the shortlisting. The first was specialization for the text-to-SQL task, since a model whose parameters already encode SQL structure can be expected to ground schema-specific queries more reliably than a general-purpose model. The second was deployability on the operator's own infrastructure, a hard requirement imposed by the privacy constraint developed in Section 2.5 that any model reachable only through a third-party cloud API is disqualified from production use on confidentiality grounds, however capable. The third

was resource footprint, since a smaller model that runs on commodity hardware lowers the barrier to on-premises adoption. Against these criteria, several open-weights models were reviewed alongside the proprietary baseline, including Meta's Llama family (Touvron et al., 2023), Mistral 7B (Jiang et al., 2023), and Alibaba's Qwen models (Bai et al., 2023), each of which offers strong general performance but is not specialized for SQL generation. Defog's SQLCoder, by contrast, is purpose-built for text-to-SQL and openly licensed for local deployment and was selected as the study's primary model; GPT-3.5 was retained as a general-purpose cloud baseline against which to measure the benefit of specialization. Table 3.2 summarizes the candidates and the rationale for including or setting aside each.

Table 3.2 — Candidate models reviewed for the study, with the rationale for inclusion or exclusion.

Model	Type	Key strength	Key weakness	Deployment	Status in this study
GPT-3.5 (OpenAI)	General-purpose LLM	Broad reasoning and fluency; simple API	Not SQL-specialized; query data leaves the network	Cloud API	Included as cloud baseline
SQLCoder-7B (Defog)	Text-to-SQL specialized	Strong schema-grounded SQL at modest size; open weights	Requires a local GPU	On-premises	Selected primary model
SQLCoder-15B (Defog)	Text-to-SQL specialized	Higher capacity than the 7B variant	Substantially heavier hardware footprint	On-premises	Considered; 7B preferred on cost/footprint
Llama (Meta)	General-purpose open LLM	Capable, widely supported, locally hostable	Not specialized for SQL; weaker structured output	On-premises	Considered; not selected
Mistral 7B (Mistral AI)	Efficient general open LLM	Strong performance per parameter; serves as a SQLCoder base	General-purpose, not SQL-tuned by default	On-premises	Considered; underlies a SQLCoder variant
Qwen (Alibaba)	General / coder open LLM	Strong coding variants; multilingual; local	Less SQL-specialized than SQLCoder	On-premises	Considered; not selected

3.6.2 Local versus Cloud Deployment

The distinction between the two selected models is not only architectural but also a matter of where the model runs, and this deployment choice carries consequences that bear directly on the research question. A cloud model such as GPT-3.5 is reached through a third-party application programming interface: the operator sends each question and, under retrieval-augmented generation, the accompanying schema and semantic-layer content — to an external provider, which performs inference on its own infrastructure and returns the result. A local model such as SQLCoder is instead downloaded as a set of open weights and executed on hardware the operator controls,

so that no query or schema ever leaves the organization’s network. The two deployment paths are contrasted in Figure 3.3.

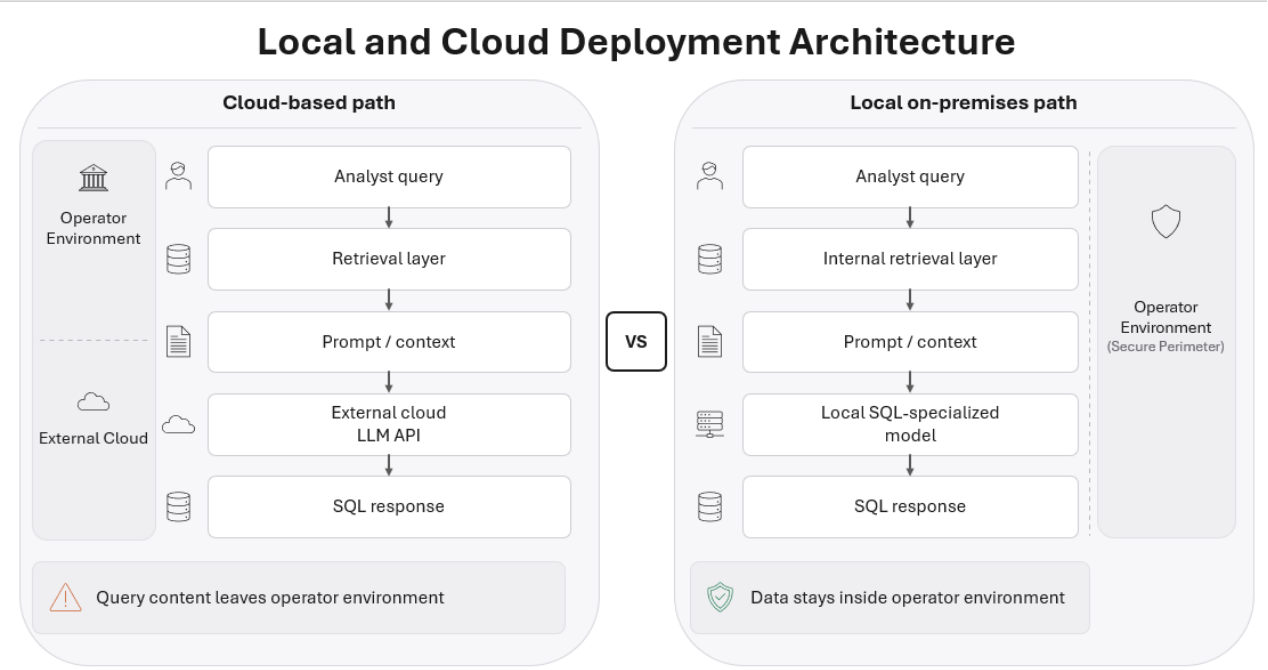

Figure 3.3 — Cloud-based versus local on-premises deployment. In the cloud path, query content leaves the operator environment, in the local path, all data stays inside the operator's secure perimeter.

Several trade-offs follow from this difference. On cost, the cloud model imposes a recurring, usage-based charge and requires no capital outlay, whereas the local model demands up-front investment in GPU hardware but no per-query fee, which favours the local option at sustained query volumes. On latency, the cloud model adds network round-trip time and is subject to provider availability and rate limits, while a local model's response time depends only on the operator's own hardware. On control, local deployment allows the operator to fix a model version, fine-tune it on proprietary data, and audit its behavior, none of which is fully possible behind a managed API.
The decisive dimension, however, is data residency. As established in Section 2.5, sending confidential subscriber and revenue data to an external provider is difficult to reconcile with confidentiality obligations under which a telecommunications operator works. A cloud API may offer superior raw capability, but for a production analytics system handling regulated customer data, the requirement that data remain within the operator's security perimeter is close to absolute. Local deployment is therefore not merely a cost or performance preference in this setting, it is the only configuration that satisfies the privacy constraint outright, and this is the central reason the study treats the cloud model as a baseline rather than a candidate for adoption.

3.7 Why These Differences Predict the Result

The three axes of difference identified above are not abstract, each maps onto one of the criteria by which the models are judged in the evaluation. Specialization bears most directly on execution success and result correctness. A model whose parameters have been tuned on tens of thousands of SQL examples is better equipped to assemble the compositionally demanding queries described in Section 3.5, the kind that require a partitioned window function to compute a year-over-year change correctly, than a general-purpose model relying on whatever SQL it happened to absorb during pre-training. The expectation, then, is that the gap between the two models will be small on simple queries and will widen as queries become structurally harder, and this is precisely

the pattern the results in Chapter 5 reveal.

The openness and deployment axis bears on privacy, which the methodology treats as an absolute rather than a tradable criterion. Because GPT-3.5 can be reached only through a cloud API, using it requires query content, potentially including personal customer data, to leave the operator's network, whereas an open model running on the operator's own hardware keeps that data resident. The architectural difference is therefore not only a question of performance but one of regulatory compliance, and for an operator acting as a data controller it can be decisive regardless of accuracy scores. The third axis, the route by which domain knowledge reaches the model, bears on the hallucination rate: a model constrained by an explicit, grounded description of the real schema has less room to invent tables or columns that do not exist than one drawing on generic pre-trained knowledge alone.

These are, at this stage, expectations rather than findings. The purpose of stating them here is to make the logic of the comparison explicit, so that the results can be read as a test of clearly articulated hypotheses rather than as an unmotivated collection of numbers. The methodology set out in the next chapter is designed to put these expectations to the test under controlled and reproducible conditions.

3.8 Chapter Summary

This chapter explained the technical concepts needed for the rest of the thesis. It defined language models, described how transformers and self-attention work, explained model training and adaptation, and showed why text-to-SQL is challenging. It also compared GPT-3.5 and SQLCoder in terms of specialisation, deployment, privacy, and expected performance. With these foundations established, Chapter 4 turns to the methodology used to build and evaluate the telecom Generative BI framework.

Chapter 4 Methodology

This chapter explains how the proposed telecom Generative BI framework was built and evaluated. It turns the technical ideas from Chapter 3 LLMs, text-to-SQL, RAG, and local deployment — into a practical seven-phase methodology. The framework moves from semantic-layer design through model selection, deployment, evaluation, and governance.

4.1 Research Design

This study follows a design-science approach. The main output is a practical artefact: a framework for building and evaluating a secure telecom Generative BI system. This fits the research aim because the thesis is not only asking whether a model performs well; it is asking how such a system can be designed responsibly for a telecom operator.

The methodology addresses the three research sub-objectives. The seven-phase framework supports the architectural objective. The model comparison and error analysis support the evaluative objective. The execution-success, correctness, runtime, hallucination, and privacy measures support the performance objective. The deployment and governance sections define the conditions under which the framework can be used safely.

4.2 Problem Framing and Target Capabilities

The methodology begins by defining the problems the system is meant to solve. Telecom BI teams often face repeated requests from stakeholders who know the business question but cannot write SQL. Data teams then become bottlenecks, and results are often delivered as raw tables rather than clear insight. Self-service analysis remains limited for non-technical users.

The target capabilities follow from these problems. The system should support natural-language querying, generate SQL, explain results in a useful way, and operate across the relevant data sources. Role-based access control is also essential because every answer must respect what the user is allowed to see.

4.3 The Seven-Phase Framework

The framework is organised into seven phases. The first three build the foundation: the semantic layer, analyst-question examples, and time-intelligence rules. The remaining phases cover model selection, deployment, evaluation, and governance. Together, these phases turn the idea of telecom Generative BI into a repeatable method.

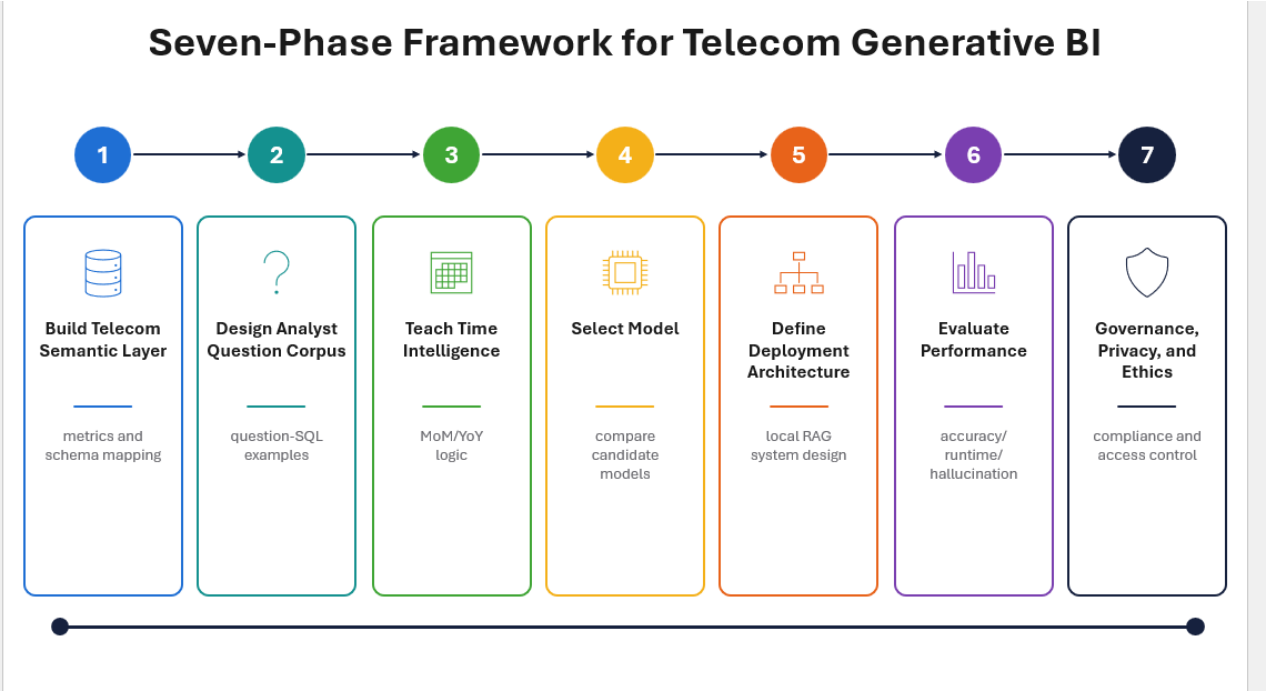

Figure 4.1 — The seven-phase framework for telecom Generative BI, from semantic-layer construction through governance.

4.3.1 Phase 1 Building the Telecom Semantic Layer

The semantic layer connects business language to the physical database schema. Without it, a model may generate SQL that runs but applies the wrong definition. For example, ARPU may be calculated incorrectly if the model does not know which revenue and subscriber rules the operator uses. This phase therefore defines the key metrics, clarifies edge cases, and maps each metric to the correct tables and columns.

Step one enumerates the core business metrics. An initial set of ten representative metrics is defined here as a worked example, with the expectation that any production deployment would extend the catalog. Table 4.1 lists those ten metrics and their business meaning.

Metric	Business Meaning
Revenue Streams	Money collected through different streams (data, voice, value-added services, etc.)
ARPU	Average revenue per user
RGS	Revenue-Generating Subscribers: subscribers who generated revenue during the period
Returnee	Subscribers who returned to the network after a period of inactivity
Churn	Lost subscribers (any subscriber not subscribed to any service for the past 90 days)
Net Adds	New + Returnee − Churn
Data Usage	MB consumed
Voice Usage	Minutes used
SMS Usage	Messages sent
Network Availability	Uptime percentage

Table 4.1 — Core telecom business metrics and definitions.

The next step is to write precise definitions for each metric and identify edge cases. For example, churn depends on how the 90-day inactivity window is defined, and returnees depend on how reactivation is treated. Each metric is then mapped to the database

objects the model must use when generating SQL.

Building on the catalogue in Table 4.1, each metric is defined in full below, with the edge-case treatment and the underlying table-and-column mappings the text-to-SQL system relies on. Operators adopting the framework should extend this catalogue with metrics specific to their reporting environment.
Revenue: Sourced from Revenue_stream.amount and Revenue_stream.date; aggregated by sum over the reporting window. Edge cases include credit notes (treated as negative entries) and inter-company revenue (excluded from external reporting).
ARPU: Computed as SUM(revenue) / COUNT(DISTINCT subscriber_id), restricted to revenue-generating subscribers within the reporting window.
RGS: COUNT(DISTINCT subscriber_id) where the subscriber generated at least one billable event in the reporting window.
Returnee: Subscribers whose previous activity occurred more than 90 days before the reporting window and who generated activity within it.
Churn: Subscribers active at the start of the reference period but inactive (no billable activity) for the subsequent 90 days.
Net Adds: New subscribers + Returnees − Churn for the reporting window.
Data Usage: SUM(usage.mb_consumed), aggregated by subscriber, region, or network element as required.
Voice Usage: SUM(usage.voice_minutes).
SMS Usage: COUNT(usage.sms_sent).
Network Availability: Element-level uptime percentage averaged across the reporting window.

4.3.2 Phase 2 Designing Analyst Question Examples

Phase 2 builds examples that reflect how analysts actually ask questions. These examples include questions such as ARPU by region, churn over the last quarter, top sites by revenue, and revenue deviation over time. The initial set contains fifty questions, which is enough for a controlled proof-of-concept while still allowing manual review. Each question is broken into five parts: the question text, the required metric, the dimensions, the time logic, and the canonical SQL. This decomposition makes the business logic explicit. It also gives the model more useful context than a simple question-and-answer pair.
The decomposed examples are stored as structured records and used as prompt context. No model parameters are updated in this study. Because the same fifty questions were also used during development and evaluation, the results should be read as internal-validation results rather than performance on unseen data.
The example below illustrates the five-part decomposition just described, together with the corresponding canonical SQL. The package is serialised as a JSON record that is used to build the schema-aware prompt context supplied to the model.
Question: ARPU by region, month over month.
Metric: ARPU = SUM(revenue) / COUNT(DISTINCT revenue-generating subscriber_id).
Dimensions: region.
Time logic: last 6 months, grouped by month.
SQL implementation:

```
SELECT
    DATE_TRUNC('month', rev.date) AS month,
    reg.region_name,
    SUM(rev.amount) / COUNT(DISTINCT s.subscriber_id) AS arpu
```

```
FROM revenue_stream rev
JOIN subscriptions sub ON rev.subscription_id = sub.subscription_id
JOIN subscribers s ON sub.subscriber_id = s.subscriber_id
JOIN regions reg ON s.region_id = reg.region_id
WHERE s.status = 'active'
    AND rev.date >= CURRENT_DATE - INTERVAL '6 months'
GROUP BY month, reg.region_name
ORDER BY month, reg.region_name;
```

JSON package:

```
{
    "persona": "analyst",
    "question": "ARPU by region month over month",
    "metric": "ARPU",
    "dimensions": ["region"],
    "time_logic": "last_6_months_MoM",
    "sql": "<see SQL implementation above>"
}
```

4.3.3 Phase 3 Time-Intelligence Handling

Time language is one of the easiest places for mistakes to enter. Phrases such as “last month,” “month to date,” and “year over year” can be interpreted in different ways. Phase 3 therefore gives the model an explicit mapping from common analyst phrases to SQL logic, reducing the chance that time-based questions are interpreted inconsistently.

Phrase	SQL Logic
last month	previous calendar month
MTD	month to date
QTD	quarter to date
MoM	current period − previous period
YoY	current period − same period prior year

Table 4.2 — Time-intelligence vocabulary and corresponding SQL logic.

Table 4.2 lists the core phrase-to-SQL mappings supplied to the model. In a production deployment this catalogue would be extended to cover quarter and year boundaries, fiscal-year variants where the operator's fiscal calendar differs from the calendar year, and rolling-window variants (“last 30 days”, “trailing twelve months”). Each phrase mapping is paired with at least one worked example in the prompt context supplied to the model.

In this proof-of-concept, time expressions are handled through prompt context rather than a separate date-resolution module. This means the model is guided by examples and phrase mappings, but it does not independently convert every phrase into an exact calendar range before writing SQL. A production system should add that capability.

The end-to-end process is simple. A user asks a question; relevant schema and example context are retrieved; that context is inserted into the prompt; the model generates SQL; the SQL is executed; and the output is evaluated. The model is therefore adapted through retrieval and prompting, not through fine-tuning.

In summary, Phase 3 adapts the pre-trained model to telecom time logic through prompt engineering and schema-aware, retrieved context, not through any parameter update or fine-tuning.

4.4 Phase 4 Model Selection: Criteria and Candidates

Model selection is based on four criteria: execution success, result correctness, runtime, and privacy. Execution success shows whether the SQL runs. Result correctness shows whether it returns the expected answer. Runtime matters for usability, but it is secondary

to correctness. Privacy is non-negotiable because telecom operators handle sensitive subscriber data and must keep it under proper control.
Two models are compared. GPT-3.5 is included as a widely used cloud baseline, even though it does not satisfy the privacy requirement for production use. SQLCoder-7B-Alpha is evaluated as the local, open-source, text-to-SQL specialized model. This comparison tests whether a smaller local model can meet both the technical and governance needs of telecom Generative BI.

4.5 Phase 5 Deployment Architecture

The methodology specifies two deployment tiers. A development and testing tier is sufficient for framework construction, training-data curation, and offline evaluation, and can be hosted on a single workstation with a quad-core CPU, 8 GB of RAM, and 1 TB of SSD storage. A production tier hosts the inference endpoint and serves analyst queries; it requires a server-class CPU, 16 to 32 GB of RAM, a GPU comparable to the RTX 3060 for inference acceleration, and 512 GB to 1 TB of SSD storage, together with a high-throughput network connection for remote users. Detailed hardware specifications are listed in Appendix D.

4.5.1 The Reference Dataset and Schema

The evaluation runs against a synthetic dataset rather than live subscriber data. This was a deliberate choice, for two reasons: it avoids any privacy or governance limit on sharing the test material, and it makes the ground truth exactly reproducible. The data is built deterministically in an in-memory DuckDB database from a fixed random seed, so every run rebuilds the same schema and the same reference results. For this proof-of-concept the dataset has four related tables: a customers table of 2,000 subscribers, each tied to one of five regions held in a small regions dimension; a monthly_usage table of 6,000 rows holding data, billing, and usage figures for each customer over three months; and a service_tickets table of 1,200 tickets. The monthly_usage and service_tickets tables link back to customers through cust_id, and customers link to regions through region_id. The full column-level schema is given in Table 4.3, and reproduced with the larger dimensional model in Appendix A.
These four tables are a cut-down version of the larger model a real deployment would use. That fuller model is a star schema built around a central fact table, FACT_TELECOM_ACTIVITY, which records activity such as revenue, data, voice, SMS, and transaction counts against shared dimensions for date, subscriber, service, region, and KPI; it is drawn as an entity-relationship diagram in Figure 4.2 and listed table by table in Appendix A (Table A.2). Keeping the evaluation schema as a clear subset of this model makes the proof-of-concept manageable while still showing how it would scale to an operator's full workload.

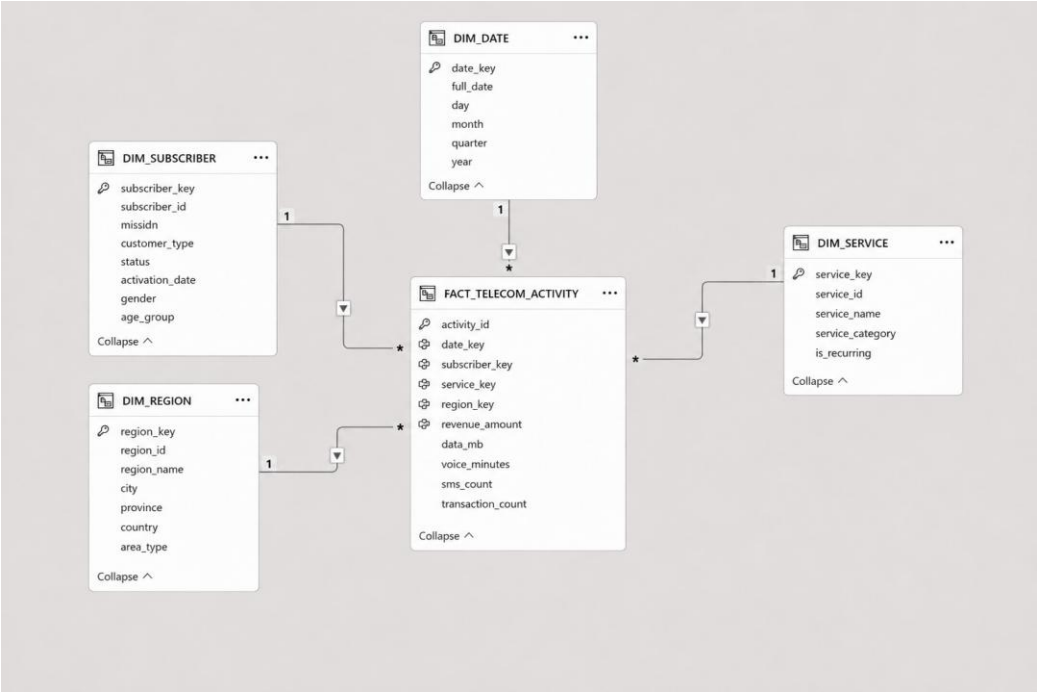

Figure 4.2 — Entity–relationship diagram of the telecom BI data schema.

This schema is also what the RAG pipeline is grounded against. The short schema string that lists each table and its columns (reproduced in Appendix A) is the context dropped into the model's prompt at inference time, and the metric definitions and time rules from the semantic layer (Section 4.3) are written in its terms. The reference SQL for every query in Table 4.5 is written and run against this same schema, so a generated query can be judged simply by comparing its result with the schema's fixed reference data.

Table	Column	Type	Description
customers	cust_id	INTEGER	Customer identifier (1000–2999); primary key
customers	plan	VARCHAR	Subscription plan: Basic, Premium, or Enterprise
customers	region_id	INTEGER	Foreign key → regions.region_id
regions	region_id	INTEGER	Region identifier (1–5); primary key
regions	region_name	VARCHAR	Region name (North, South, East, West, Central)
monthly_usage	cust_id	INTEGER	Foreign key → customers.cust_id
monthly_usage	month_num	INTEGER	Month index (1, 2 or 3)
monthly_usage	gb_used	DOUBLE	Data consumed in the month, GB (range 5–100)
monthly_usage	bill	DOUBLE	Monthly bill amount (range 20–150)
service_tickets	ticket_id	INTEGER	Ticket identifier (50000–51199); primary key
service_tickets	cust_id	INTEGER	Foreign key → customers.cust_id
service_tickets	priority	VARCHAR	Ticket priority: Low,

Table	Column	Type	Description
			Medium, or High

Table 4.3 — Schema of the synthetic reference database used for evaluation (the four-table proof-of-concept implementation).

4.5.2 Implementation of the RAG-Based SQL Generation Pipeline

The pipeline turns a natural-language question into SQL through retrieval-augmented prompting rather than fine-tuning, so the pre-trained model is never changed; it is simply given the right database context at the moment it is asked. The flow is straightforward. A question comes in, the system gathers the schema of the active dataset and a few relevant SQL patterns from a small knowledge base and builds these into a prompt. The model generates a query, which is then run against the database. If it fails, the model is asked to fix it; once it runs, the system records whether it executed and whether the result matched the expected answer.

Input handling:

The question is never sent to the model on its own. It always travels with schema and example context, so the SQL that comes back is built around tables and columns that actually exist rather than ones the model has imagined.

Schema and context:

For each question, the system assembles two types of contexts. First, it extracts the active dataset schema that contains tables, columns, and data types and adds it to the prompt as a concise description. Second, it retrieves relevant worked examples from a small library of question-and-SQL pairs. These examples are embedded with the all-MiniLM-L6-v2 sentence-transformer, stored in a FAISS index, and matched to the current question. They guide the model through examples rather than training, so the model's parameters stay unchanged. This approach is lighter than retrieving from a full operator document store: it combines schema-aware prompting with targeted semantic search over known SQL patterns.

Prompt construction:

The schema and the retrieved patterns are dropped into a fixed template alongside the question. Its structure is:

```
You are an expert BI data analyst.
Database schema:
[table and column definitions for the active dataset]
Helpful Patterns:
[example questions and their SQL, retrieved from the knowledge base]
Task: [the user's natural-language question]
[feedback from a previous failed attempt, if any]
Rules:
1. Return only valid DuckDB SQL inside a single ```sql block.
2. Do not add explanations.
3. Use the exact column names from the schema, quoting any with special characters.
4. Prefer USING() over ON when joining.
```

The wording shifts with the question and the patterns that come back, but the skeleton is always the same.

Model call:

The finished prompt goes to Defog SQLCoder-7B-Alpha, running locally through Ollama, and the model returns its query inside a ```sql block. At no point are any weights updated every bit of task-specific behaviour comes from the prompt, and the model itself is the

same from the first query to the last.

Post-processing and self-healing:
The SQL is lifted out of the ```sql block. If it will not run, the system does not give up on the first try: it hands the model back its own query together with the database's error message and asks for a fix, and it will do this up to three times. That loop clears the easy syntax and schema slips, but it is not a real SQL-repair tool — anything still broken after three attempts is logged as an execution failure.

Execution and logging:
Each query is run against the in-memory DuckDB database, and two things are written down: whether it ran at all (execution success) and whether its result matched the expected answer (result correctness). Those two records are what Chapter 5 reports.

Time-related queries:
There is no dedicated module for resolving dates. Relative phrases like “last month,” “current year,” or “previous quarter” are left to the prompt and the model's own reading of the question, helped by the time vocabulary from Section 4.3.3. That is a genuine weakness: these phrases stay ambiguous until they are pinned to actual dates. A stronger version would resolve them first — turning “last month” into a concrete start and end date based on the run date — and put those dates in the prompt before the model ever sees the question.

RAG versus fine-tuning:
The two are easy to confuse but work differently. Fine-tuning rewrites a model's parameters with extra training data; the approach here leaves the pre-trained model alone and feeds it the context it needs as it runs. The honest description of this system is therefore a pre-trained model driven by retrieval-augmented prompting, not a fine-tuned one.

Limitations of the approach:
Retrieval-augmented prompting sharpens the context the model sees, but it does not fix everything. The model can still write wrong SQL when the retrieved context is thin, ambiguous, or too broad. Because it is never fine-tuned, the quality of its output rides almost entirely on the prompt and on how clearly the schema is laid out. Apart from the rules in the prompt and the self-healing retry, nothing strictly stops it from inventing a table or column name. And, as already noted, relative time expressions are not handled reliably without a proper date-resolution step.

4.6 Phase 6 Evaluation Protocol

The evaluation protocol measures more than whether a query runs. Each model is tested on execution success, result correctness, runtime, hallucination, and privacy. The fifty-question set is divided into simple, moderate, and complex questions so that performance can be understood by difficulty level, not only as a single total score.

4.6.1 Query Set and Difficulty Taxonomy

The set comprises fifty natural-language questions, each paired with a reference SQL query and an expected result set computed directly from the schema and reference data. Fifty items is a deliberate compromise. It is large enough that a result-correctness figure is not dominated by the coarse percentage steps a very small test imposes — a three-

item test can only land on 0, 33, 67, or 100 percent — yet small enough that every query and every generated answer can still be inspected by hand, preserving the qualitative depth that a purely automated benchmark sweep would lose. The questions were drawn from the recurring information needs of telecom business intelligence teams identified in Section 4.2, and the full set was fixed in advance of any model being run, so that neither model could be favored by selecting questions after the fact.

To expose how performance varies with difficulty, the set is stratified into three bands: twenty simple, fifteen moderate, and fifteen complex questions defined by the structural demands that the correct SQL places on the model. A simple query targets a single table and requires at most one aggregation or filter, with no joins and no time logic. A moderate query requires two or three tables to be joined, together with a grouping or aggregation and at most one straightforward time filter. A complex query demands three or more joins, partitioned aggregations or window functions, and typically period-over-period time intelligence such as a year-over-year comparison or a nested subquery. The worked examples used earlier in this chapter map onto this scheme: revenue per region is a moderate query, whereas revenue per region year over year, which requires a partitioned window function, is complex. Table 4.4 summarizes the taxonomy and the composition of the set.

Difficulty band	Defining SQL characteristics	Example analyst question	Count
Simple	Single table, at most one aggregation or filter, no joins, no time logic	How many active subscribers are there this month?	20
Moderate	Two or three joined tables; a grouping or aggregation; at most one simple time filter	What is the revenue per region?	15
Complex	Three or more joins; partitioned aggregations or window functions; period-over-period time intelligence	What is the revenue per region year over year?	15

Table 4.4 — Query difficulty taxonomy and the composition of the fifty-question evaluation set.

The full set is listed in Table 4.5 (three representative examples; the full set of fifty questions is given in Appendix E) below. The twenty simple questions exercise single-table retrieval and basic aggregation; the fifteen moderate questions introduce joins across the three tables together with grouping and HAVING filters; and the fifteen complex questions require window functions, period-over-period comparison, or pivoting. Each question is paired with a reference SQL query and an expected result set computed directly against the reference schema; the reference queries are reproduced with the evaluation code in the appendices.

#	Difficulty	Natural-language question	Key SQL features
1	Simple	Count the total number of customers.	Single table; COUNT
21	Moderate	Count the number of tickets for each plan.	Join customers–tickets; GROUP BY, COUNT
45	Complex	Customer with the largest month-over-month gb_used	LAG; ordering

#	Difficulty	Natural-language question	Key SQL features
		increase.	

Table 4.5 — Three representative evaluation queries (one per difficulty band). The complete fifty-question set is listed in Appendix E (Table E.1).

4.6.2 Metrics and Their Measurement

Execution Success Rate:

The execution success rate measures whether the SQL a model generates runs against the reference database without raising a syntax or runtime error. Each of the fifty generated queries is executed once and counts as a success if it completes and returns a result set, regardless of whether that result is the one the question intended. The rate is reported as the proportion of the fifty queries that execute successfully, overall and within each difficulty band. Because it ignores whether the answer is right, execution success is a necessary but not sufficient condition for a correct answer: a query can execute successfully and still be wrong, which is exactly what the result-correctness metric below captures. Each figure is reported as a simple proportion, without confidence intervals or other inferential statistics, for the reasons given in Section 4.6.3.

Result Correctness:

Result correctness measures whether the executed SQL query returns the expected output for a given natural-language question. A query that executes successfully (the execution-success criterion above) can still fail this test if the result set it returns does not match the predetermined expected answer; result correctness is therefore the stricter of the two checks, and it is formalized in Equation (4.1) below. It is reported as the proportion of the fifty queries whose result matches the expected answer, overall and within each difficulty band. It does not inspect the internal structure of the SQL — the joins, clause ordering, or window functions used — because the harness compares executed results, not query form.

$$RC = \left(\frac{\text{number of queries whose result set matches the reference result set}}{N} \right) \times 100\%$$

where N = total number of test queries

(4.1)

Equation (4.1) formalizes result correctness: RC = (100 / N) × Σ I(R_i = $\hat{R}_i$), where RC is result correctness, N is the number of evaluated queries, R_i is the reference result set and $\hat{R}_i$ is the result set returned by the generated query for query i, and I(·) is an indicator equal to 1 when the two result sets match and 0 otherwise (Li et al., 2023).

Runtime (End-to-End Latency):

Runtime is measured end-to-end, from the moment a question is submitted to the moment its result is returned, on the hardware specified in Section 4.5. Runtime was

recorded as observed end-to-end response time and summarized by difficulty band. Because repeated runtime trials were not systematically performed, the runtime values should be interpreted as indicative deployment measurements rather than stable performance benchmarks. Runtime is treated as a secondary criterion (4.4): an analyst will tolerate a few extra seconds for a correct answer, so runtime informs but does not decide model selection.

Hallucination Rate (Schema Faithfulness):

In a text-to-SQL setting, a hallucination is SQL that references columns, tables, or relationships that do not exist in the telecom schema, or that fabricates a result not grounded in the underlying data. This is the structured-query analogue of the hallucination problem discussed in 2.4. Evaluation is conducted by inspection: each generated query is cross-referenced against the semantic layer (4.3.1), and any element not present in, or reasonably derivable from, the schema is categorized as a hallucination. The hallucination rate is reported as the proportion of the fifty generated queries containing one or more such unsupported elements. A grounded semantic layer is expected to drive this rate down by constraining the model to the operator's actual schema, distinguishing the RAG-based approach from a baseline model relying on pre-trained knowledge alone.

$$\text{Hallucination Rate} = \frac{\text{number of queries containing unsupported schema elements or ungrounded logic}}{N} \times 100\%$$

where N = total number of test queries

(4.2)

where a schema-ungrounded element is any table, column, or relationship that is absent from the telecom schema, and N is the number of generated queries (Ji et al., 2023).

4.6.3 Reporting and Interpretation

The results are reported as descriptive internal-validation figures. They are useful for comparing the two models within this proof-of-concept setup, but they should not be treated as statistical estimates for all telecom environments. A production operator would need to repeat the evaluation using its own schema, business definitions, and query set. The numerical results are reported in Chapter 5. They are this study's own measurements, obtained by running the protocol defined here against the reference schema and the fifty-question set. Where figures from published benchmark evaluations are cited, they are identified explicitly as external results and used only for context, not merged into this study's own measurements. As a proof-of-concept conducted on a representative schema rather than a live production estate, the evaluation is not a claim about every telecom database: an operator wishing to act on these results would still need to re-evaluate against its own schema and query set, because no result on one schema guarantees a corresponding result on another.
McNemar's test is used for paired binary outcomes. It is appropriate when two methods are evaluated on the same cases and each case is classified as correct or incorrect. In this study it is appropriate because both GPT-3.5 and SQLCoder were evaluated on the

same 50 queries, and each query is marked result-correct or result-incorrect for each model. The test looks only at the disagreement cases as example the queries where one model was correct and the other was not and ignores the queries on which both models agree. Because the number of disagreements here is small, the exact form of the test is used rather than the chi-squared approximation.

4.7 Phase 7 Ethics, Data Governance, and Privacy

Because telecom operators act as data controllers, privacy and governance are central to the methodology. The most important requirement is locality: model inference should take place inside infrastructure controlled by the operator, rather than sending sensitive data to an external API.
Access control and auditability are also essential. The system must only return data the user is authorized to see, and it must log queries and outputs so that results can be reviewed and investigated if needed.
A local open-source model supports the locality requirement. Access control and audit logging, however, must be implemented by the surrounding application and governance layer. The model alone does not solve those responsibilities.

Chapter 5 Results and Experiment

5.1 Introduction

This chapter compares GPT-3.5 and SQLCoder-7B-Alpha against the evaluation criteria defined in Chapter 4. GPT-3.5 represents a general-purpose cloud model, while SQLCoder-7B-Alpha represents a specialized model that can run locally. The aim is to show not only which model performs better, but also why each model succeeds or fails. The evaluation uses the fifty-question set described in Chapter 4. The results are reported across execution success, result correctness, runtime, hallucination, and privacy. Three representative queries are examined more closely to make the model's

behaviour easier to interpret: one simple, one moderate, and one complex.

5.2 Execution Success Rate

Execution success measures whether the generated SQL runs without error. GPT-3.5 executed 42 of the 50 queries successfully, or 84 percent. SQLCoder-7B-Alpha executed 45, or 90 percent. Both models handled the representative simple and moderate queries successfully, and both produced runnable SQL for the complex query. However, execution alone does not prove correctness: GPT-3.5's complex query ran but returned the wrong result because it missed the required year-over-year window logic.

Query	Difficulty	GPT-3.5	SQLCoder-7B-Alpha
Revenue YoY %	Simple	Pass	Pass
Revenue per region	Moderate	Pass	Pass
Revenue per region YoY	Complex	Executes (result incorrect; see 5.3)	Pass
Execution success rate (all 50 queries)	—	84% (42/50)	90% (45/50)

Table 5.2 — Execution success (whether each query ran without error) for three representative queries and the aggregate over all fifty. Result correctness is reported separately in Section 5.3.

5.3 Result Correctness

Result correctness is the stricter test because it checks whether the executed SQL returns the expected answer. Both models answered the representative simple and moderate queries correctly. The difference appeared in the complex query: GPT-3.5 produced a runnable but incorrect query, while SQLCoder-7B-Alpha returned the correct result. Across all fifty questions, GPT-3.5 achieved 72 percent correctness and SQLCoder-7B-Alpha achieved 84 percent.

Metric	GPT-3.5	SQLCoder-7B-Alpha
Result correctness (all 50 queries)	72% (36/50)	84% (42/50)

Table 5.3 — Aggregate result correctness across the set.

Because both models were evaluated on the same 50 queries, their per-query outcomes can be compared directly. The paired result-correctness outcomes are cross-tabulated in the contingency table below.

	SQLCoder correct	SQLCoder incorrect	Row total
GPT-3.5 correct	33	3	36
GPT-3.5 incorrect	9	5	14
Column total	42	8	50

Table 5.3b — Paired result-correctness outcomes for GPT-3.5 and SQLCoder-7B-Alpha across the 50 queries.

The contingency table shows that both models returned the correct result on 33 of the 50 queries. GPT-3.5 alone was correct on 3 queries, while SQLCoder-7B-Alpha alone was correct on 9. Both models were incorrect on the remaining 5 queries.
The two disagreement cells define b = 3 (GPT-3.5 correct, SQLCoder incorrect) and c =

9 (GPT-3.5 incorrect, SQLCoder correct), so the total number of disagreements is b + c = 12. Because this count is small, McNemar's exact test is used rather than the chi-squared approximation. McNemar's exact test produced a two-sided p-value of approximately 0.146. The paired comparison descriptively favors SQLCoder, since SQLCoder alone returned the correct result on 9 queries compared with 3 queries for GPT-3.5 alone. However, McNemar's exact test was not statistically significant at the 5% level. Therefore, the result should be interpreted as descriptive support within this 50-query proof-of-concept evaluation, not as conclusive statistical evidence of general model superiority.

5.4 End-to-End Response Time

End-to-end response time was similar for both models in this setup. Simple queries took about 5 seconds, moderate queries about 8 seconds, and complex queries about 15 seconds. Since neither model had a meaningful runtime advantage, runtime does not decide the comparison. Correctness and privacy matter more.

Difficulty band	GPT-3.5 median (s)	SQLCoder-7B median (s)
Simple	5	5
Moderate	8	8
Complex	15	15

Table 5.4 — End-to-end runtime by difficulty band, in seconds

These timings should be read as practical response-time measurements, not as a pure benchmark of model speed. SQLCoder was run locally, while GPT-3.5 was accessed through an API. The measured time therefore includes different factors such as hardware performance, network latency, and external server processing.
Runtime was logged as end-to-end response time, but only per-band median values were recorded. The full per-query distribution — mean, standard deviation, minimum, and maximum, is not available for this proof-of-concept.
A production evaluation should log per-query timings so that this variability can be reported, and should separate model inference time, retrieval time, SQL execution time, and any network overhead.

5.5 Hallucination

In this study, hallucination means that a model uses a table, column, relationship, or result logic that is not grounded in the schema. SQLCoder-7B-Alpha showed no hallucination in the representative cases. GPT-3.5 did not invent a table or column, but it lost alignment with the required logic in the complex year-over-year query by omitting the per-region partition. This produced a fluent and executable but incorrect answer.

Model	Hallucinations (of 3 categories)	Detail
GPT-3.5	1	Complex query: omitted the PARTITION BY clause in the year-over-year window function.
SQLCoder-7B-Alpha	0	None across all three categories

Table 5.5 — Hallucination outcomes across the three representative queries.

Error category	GPT-3.5	SQLCoder-7B
Schema hallucination (nonexistent table/column)	3	0
Incorrect or missing join	0	1
Wrong or missing aggregation	0	2
Missing or incorrect time logic	1	0
Incorrect telecom-metric definition	1	0
Syntax error (failed to execute)	9	5

Table 5.6 — Error analysis by category across the full 50-query evaluation set.

5.6 Privacy and Security

Privacy is the decisive criterion. GPT-3.5 requires inference through external infrastructure, which means query content may leave the environment the operator controls. For telecom data, this is a serious governance issue. SQLCoder-7B-Alpha can run locally, so the operator can keep schema, query content, and sensitive data inside its own infrastructure. This makes SQLCoder the only viable production option in this privacy-constrained setting.

5.7 Summary

Overall, SQLCoder-7B-Alpha performed better on execution success and result correctness, matched GPT-3.5 on practical runtime, and satisfied the privacy requirement. GPT-3.5 remains useful as a cloud baseline, but it is not suitable for production use where subscriber data and internal analytics must remain inside the operator's environment.

Criterion	GPT-3.5 (API)	SQLCoder-7B (on-premises, pre-trained)
Execution success rate	84% (42/50)	90% (45/50)
Simple Category — revenue YoY %	Pass	Pass
Moderate Category — revenue per region	Pass	Pass
Complex Category — revenue per region YoY	Incorrect result (omits window function)	Pass
Aggregate result correctness	72% (36/50)	84% (42/50)
Hallucination rate	Hallucinated on the Complex category (per-region YoY)	None across all three categories
Runtime: simple queries	5 s	5 s
Runtime: mid-complexity	8 s	8 s
Runtime: complex queries	15 s	15 s
Privacy	Fails: inference on third-party infrastructure	Satisfies: fully local deployment feasible

Table 5.7 — Model comparison across execution success rate, result correctness, runtime, hallucination, and privacy. Execution success rate and result correctness are internal-validation figures: the same fifty queries were used during development and evaluation, so they are not estimates of performance on unseen queries. Runtime values are the per-band medians.

Chapter 6 Recommendations and Conclusion

6.1 Key Findings

The results point to several clear insights.

In this proof-of-concept evaluation, the locally deployed SQL-specialized model performed better than the cloud baseline on execution success and result correctness, while also satisfying the privacy requirement. The gap between the two models was one of reasoning rather than syntax: only the complex year-over-year query separated them. Hallucination affected only the cloud model — on the complex year-over-year category, GPT-3.5 produced ungrounded query logic, omitting the PARTITION BY and the window function the comparison required, while SQLCoder-7B hallucinated simple queries and used GROUP BY instead. Speed did not prove decisive, as both models answered in roughly 5 to 15 seconds depending on complexity, whereas privacy did: GPT-3.5 fails the data-residency requirement regardless of its scores. These figures are nonetheless provisional, reflecting internal validation on a fifty-question set rather than unseen-query performance, and their significance is limited — McNemar's exact test on the 50 queries (two-sided $p \approx 0.146$) makes SQLCoder's correctness advantage descriptive, not statistically significant at the 5% level.

6.2 Recommendations for Telecom Operators

Based on the findings, telecom operators should prioritize several actions when deploying Generative BI.

Operators should run a local, open-weights model, because it beats the cloud baseline on both metrics 90% vs 84% execution, 84% vs 72% result-correctness, while keeping data in-house. They should build the semantic layer first, since grounding mattered more than model size, and adapt the model with retrieval-augmented prompting rather than fine-tuning. Data residency should be treated as a hard line, as it overrode every technical score. Finally, the three metrics should be scored separately and re-tested on an unseen set before the numbers are trusted.

6.3 Deployment Priorities

The semantic layer should be built before investing in larger models: a clear definition of ARPU, churn, revenue, and time comparisons stops even a strong model from generating confident but incorrect SQL. Privacy and data residency should be designed into the architecture from the start, since a local model supported by RAG avoids the governance risk of sending query content to an external endpoint.

6.4 Future Work

Future work should test the framework on a larger telecom schema and a separate unseen test set to see whether the results generalize beyond the development set. The sample size should be increased by extending from commercial reporting to network analytics KPIs — for example, availability, accessibility, throughput, dropped calls, and traffic — which would provide more queries and allow more statistical tests across more than 50 queries and more than two models. Production controls should also be added: role-based access at the natural-language layer, audit logging, human review of high-impact outputs, and explicit time normalization for relative phrases such as “last month” or “year to date.” Finally, this was the pilot phase, and the framework is now being validated; in the scaling phase that follows the machine-learning models will be

embedded within it so that it does not just return insights or tables but delivers real predictions using a combined ARIMA and XGBoost model or classifies the data using clustering.

6.5 Limitations

This study is a proof-of-concept, not a production benchmark. The set contains fifty questions, which supports close inspection but limits statistical generalization. The same questions were used during development and evaluation, so the results are internal-validation figures rather than estimates on unseen questions. The semantic layer and analyst-question set are also illustrative, not production complete.
The runtime results also have limits because the models were deployed differently: SQLCoder ran locally, while GPT-3.5 used an external API. The timings therefore reflect user-experience latency in this setup, not a pure model-speed comparison.

6.6 Conclusion

With the right architecture, telecom operators can use Generative BI to make analytics more accessible without giving up privacy or control. The strongest path identified in this thesis is a locally deployed, text-to-SQL-specialized model supported by a telecom semantic layer and RAG. In telecom, trustworthy AI is not only about intelligence. It is about correctness, governance, and data control.

References

1. Affolter, K., Stockinger, K. and Bernstein, A. (2019) 'A comparative survey of recent natural language interfaces for databases', The VLDB Journal, 28(5), pp. 793–819. doi:10.1007/s00778-019-00567-8.
2. Anodot (2025) Autonomous monitoring for telecom virtualized environments. Available at: https://www.anodot.com/blog/network-monitoring-telecom-virtualized-environments/ (Accessed: 16 June 2026).
3. AT&T (2023) AT&T's new generative AI tool will help support employees (Ask AT&T). Available at: https://about.att.com/blogs/2023/generative-ai.html (Accessed: 16 June 2026).
4. Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., et al. (2023) 'Qwen technical report', arXiv preprint arXiv:2309.16609.
5. Bengio, Y., Simard, P. and Frasconi, P. (1994) 'Learning long-term dependencies with gradient descent is difficult', IEEE Transactions on Neural Networks, 5(2), pp. 157–166.
6. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020) 'Language models are few-shot learners', Advances in Neural Information Processing Systems, 33, pp. 1877–1901.
7. Deutsche Telekom (2024) Global Telco AI Alliance founding parties sign agreement to establish a joint venture focused on co-developing and launching a multilingual Telco LLM. Available at: https://www.telekom.com/en/media/media-information/archive/agreement-jv-for-telco-specific-llm-1068392 (Accessed: 16 June 2026).
8. Holtzman, A., Buys, J., Du, L., Forbes, M. and Choi, Y. (2020) 'The curious case of neural text degeneration', International Conference on Learning Representations (ICLR).
9. IBM (2025) What is retrieval-augmented generation (RAG)? Available at: https://www.ibm.com/think/topics/retrieval-augmented-generation (Accessed: 16 June 2026).
10. Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y.J., Madotto, A. and Fung, P. (2023) 'Survey of hallucination in natural language generation', ACM Computing Surveys, 55(12), pp. 1–38.
11. Jiang, A.Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D.S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. (2023) 'Mistral 7B', arXiv preprint arXiv:2310.06825.
12. Johnson, J., Douze, M. and Jégou, H. (2019) 'Billion-scale similarity search with GPUs', IEEE Transactions on Big Data, 7(3), pp. 535–547.
13. Kaplan, J., McCandlish, S., Henighan, T., Brown, T.B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J. and Amodei, D. (2020) 'Scaling laws for neural language models', arXiv preprint arXiv:2001.08361.
14. Karpukhin, V., Oğuz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D. and Yih, W.-t. (2020) 'Dense passage retrieval for open-domain question answering', Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 6769–6781.
15. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S. and Kiela, D. (2020) 'Retrieval-augmented generation for knowledge-intensive NLP tasks', Advances in Neural Information Processing Systems, 33, pp. 9459–9474.
16. Li, J., Hui, B., Qu, G., Yang, J., Li, B., Li, B., Wang, B., Qin, B., Geng, R., Huo, N., et al. (2023) 'Can LLM already serve as a database interface? A big bench for large-scale database grounded text-to-SQLs', Advances in Neural Information Processing Systems, 36.
17. Mikolov, T., Sutskever, I., Chen, K., Corrado, G. and Dean, J. (2013) 'Distributed representations of words and phrases and their compositionality', Advances in Neural Information Processing Systems, 26, pp. 3111–3119.
18. Panintelligence (2025) Traditional BI vs. Generative BI: Understanding the Shift. Available at: https://panintelligence.com/blog/traditional-bi-vs-generative-bi/ (Accessed: 13 June 2026).
19. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. and Sutskever, I. (2019) 'Language models are unsupervised multitask learners', OpenAI Technical Report.
20. Sennrich, R., Haddow, B. and Birch, A. (2016) 'Neural machine translation of rare words with subword units', Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL), pp. 1715–1725.
21. SK Telecom (2024) Global Telco AI Alliance (GTAA): founding parties sign joint-venture agreement to co-develop a multilingual Telco LLM. Available at: https://www.sktelecom.com/en/press/press_detail.do?idx=1612 (Accessed: 16 June 2026).

22. Techplayon (2025) What is RAG and how it is transforming telecom operations? Available at: https://www.techplayon.com/what-is-rag-and-its-use-in-telecom-operations/ (Accessed: 16 June 2026).
23. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. (2023) 'Llama 2: open foundation and fine-tuned chat models', arXiv preprint arXiv:2307.09288.
24. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I. (2017) 'Attention is all you need', Advances in Neural Information Processing Systems, 30, pp. 5998–6008.
25. Vodafone (2024) Meet SuperTOBi: Vodafone's new generative AI virtual assistant, now serving customers in multiple countries. Available at: https://www.vodafone.com/news/newsroom/technology/meet-super-tobi-vodafone-s-new-generative-ai-virtual-assistant-now-serving-customers-in-multiple-countries (Accessed: 16 June 2026).
26. Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E.H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J. and Fedus, W. (2022) 'Emergent abilities of large language models', Transactions on Machine Learning Research.
27. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z. and Radev, D.R. (2018) 'Spider: a large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task', Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. Brussels: Association for Computational Linguistics, pp. 3911–3921.

Appendix A — Database Schema

The evaluation in Chapter 4 is conducted against a synthetic reference dataset generated deterministically, with a fixed random seed, in an in-memory DuckDB database, so that the schema and the reference result sets are exactly reproducible. The dataset comprises four related tables. A small regions dimension table enumerates five geographic regions; the customers table holds 2,000 subscribers, each assigned to one region through a region_id foreign key; the monthly_usage table holds 6,000 rows, one per customer for each of three months; and the service_tickets table holds 1,200 support tickets. Both monthly_usage and service_tickets reference customers through the cust_id foreign key. Table A.1 details the schema.

Table	Column	Type	Description
customers	cust_id	INTEGER	Customer identifier (1000–2999); primary key
customers	plan	VARCHAR	Subscription plan: Basic, Premium, or Enterprise
monthly_usage	cust_id	INTEGER	Foreign key → customers.cust_id
monthly_usage	month_num	INTEGER	Month index (1, 2 or 3)
monthly_usage	gb_used	DOUBLE	Data consumed in the month, GB (range 5–100)
monthly_usage	bill	DOUBLE	Monthly bill amount (range 20–150)
service_tickets	ticket_id	INTEGER	Ticket identifier (50000–51199); primary key
service_tickets	cust_id	INTEGER	Foreign key → customers.cust_id
service_tickets	priority	VARCHAR	Ticket priority: Low, Medium, or High
customers	region_id	INTEGER	Foreign key → regions.region_id
regions	region_id	INTEGER	Region identifier (1–5); primary key
regions	region_name	VARCHAR	Region name (North, South, East, West, Central)

Table A.1 — Schema of the synthetic reference database used for evaluation.

The schema string supplied to the model at inference time is: customers(cust_id, plan, region_id), regions(region_id, region_name), monthly_usage(cust_id, month_num, gb_used, bill), service_tickets(ticket_id, cust_id, priority). This compact form is the grounding context that the retrieval-augmented pipeline (Section 3.5.1) inserts into the prompt.

The broader star-schema data model depicted in Figure 4.2 represents a conceptual target model for a full production deployment. It organises telecom activity as a central fact table surrounded by conformed dimensions, as set out in Table A.2. The proof-of-concept evaluation in this thesis is conducted against the simplified four-table subset

documented in Table A.1 above.

Table	Column	Key	Description
dim_date	date_key	PK	Surrogate key for the calendar date
dim_date	full_date		Calendar date
dim_date	day		Day of the month
dim_date	month		Month of the year
dim_date	quarter		Calendar quarter
dim_date	year		Calendar year
dim_subscriber	subscriber_key	PK	Surrogate key for the subscriber
dim_subscriber	subscriber_id		Natural subscriber identifier
dim_subscriber	msisdn		Mobile number (MSISDN)
dim_subscriber	customer_type		Customer type (e.g. prepaid, postpaid)
dim_subscriber	status		Subscriber status (active, inactive)
dim_subscriber	activation_date		Date the subscriber was activated
dim_subscriber	gender		Subscriber gender
dim_subscriber	age_group		Age band
dim_service	service_key	PK	Surrogate key for the service
dim_service	service_id		Natural service identifier
dim_service	service_name		Service name
dim_service	service_category		Service category (voice, data, SMS, VAS)
dim_service	is_recurring		Whether the service is recurring
dim_region	region_key	PK	Surrogate key for the region
dim_region	region_id		Natural region identifier
dim_region	region_name		Region name
dim_region	city		City
dim_region	province		Province or state
dim_region	country		Country
dim_region	area_type		Area type (e.g. urban, rural)
dim_kpi	kpi_key	PK	Surrogate key for the KPI
dim_kpi	kpi_name		KPI name (e.g. ARPU, churn)
dim_kpi	business_definition		Business definition of the KPI
dim_kpi	formula		Calculation formula
dim_kpi	calculation_grain		Grain at which the KPI is computed
fact_telecom_activity	activity_id	PK	Surrogate key for the activity record
fact_telecom_activity	date_key	FK	Refer to the dim_date.date_key
fact_telecom_activity	subscriber_key	FK	Refers to the dim_subscriber.subscriber_key
fact_telecom_activity	service_key	FK	Refers to the dim_service.service_key
fact_telecom_activity	region_key	FK	Refers to the dim_region.region_key
fact_telecom_activity	revenue_amount		Revenue generated by the activity
fact_telecom_activity	data_mb		Data consumed (MB)
fact_telecom_activity	voice_minutes		Voice usage (minutes)
fact_telecom_activity	sms_count		Number of SMS messages
fact_telecom_activity	transaction_count		Number of transactions

Table A.2 — Logical data model of the telecom BI star schema shown in Figure 4.2 (conceptual target model).

Appendix B — Application Code: Retrieval-Augmented Generation Agent

The following listing reproduces the Streamlit application implementing the retrieval-augmented text-to-SQL agent: Defog SQLCoder-7B served locally through Ollama, FAISS retrieval over a small SQL-pattern knowledge base, dynamic schema extraction from a connected DuckDB database file, and a self-healing loop that re-prompts the model with the DuckDB error message when a generated query fails to execute.

```
import streamlit as st
import pandas as pd
import duckdb
import uuid
```

```
import re
import os

# --- Updated imports (old ones are deprecated) ---
try:
    from langchain_ollama import OllamaLLM
except ImportError:
    # Fallback for older installs; tell the user to upgrade
    from langchain_community.llms import Ollama as OllamaLLM  # type: ignore

try:
    from langchain_huggingface import HuggingFaceEmbeddings
except ImportError:
    from langchain_community.embeddings import HuggingFaceEmbeddings  # type: ignore

from langchain_core.prompts import PromptTemplate
from langchain_community.vectorstores import FAISS
from langchain_core.documents import Document

# ========================================
# UI & Page Config
# ========================================
st.set_page_config(page_title="Dynamic BI Agent", page_icon="", layout="wide")

# ========================================
# 1. MEMORY SETUP FOR MULTIPLE CHATS
# ========================================
if "chats" not in st.session_state:
    first_chat_id = str(uuid.uuid4())
    st.session_state.chats = {first_chat_id: []}
    st.session_state.current_chat_id = first_chat_id

# ========================================
# 2. CACHED HEAVY RESOURCES
# ========================================
@st.cache_resource
def load_llm():
    # Defog SQLCoder 7B with extended context for big schemas + self-healing prompts
    return OllamaLLM(
        model="sqlcoder:7b",
        temperature=0.01,
        num_ctx=8192,
        num_predict=2048,
    )

@st.cache_resource
def load_rag_memory():
    embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

    sql_knowledge_base = [
        {"task": "Rank items based on a sum within a group.",
         "sql": "SELECT id, category, RANK() OVER(PARTITION BY category ORDER BY sum_val DESC) FROM (SELECT
id, category, SUM(val) AS sum_val FROM my_table GROUP BY 1, 2);"},
        {"task": "Pivot rows into columns based on a condition.",
         "sql": "SELECT id, SUM(CASE WHEN type = 'A' THEN val ELSE 0 END) AS A_val FROM my_table GROUP BY
id;"},
        {"task": "Filter out the top and bottom percentiles.",
         "sql": "WITH ranked AS (SELECT id, PERCENT_RANK() OVER(ORDER BY val DESC) AS pr FROM my_table)
SELECT id FROM ranked WHERE pr <= 0.05;"},
        {"task": "Show a running total or cumulative sum across time.",
         "sql": "SELECT id, month, SUM(val) OVER(PARTITION BY id ORDER BY month) FROM my_table;"},
        {"task": "Calculate the difference between current row and previous row (Lag).",
         "sql": "SELECT id, time, val - LAG(val) OVER(PARTITION BY id ORDER BY time) FROM my_table;"},
        {"task": "Find percentage of total across all groups.",
         "sql": "SELECT category, SUM(val) * 100.0 / SUM(SUM(val)) OVER() FROM my_table GROUP BY
category;"},
    ]

    docs = [Document(page_content=ex["task"], metadata={"sql": ex["sql"]}) for ex in sql_knowledge_base]
    vectorstore = FAISS.from_documents(documents=docs, embedding=embeddings)
    return vectorstore.as_retriever(search_kwargs={"k": 2})

# ========================================
# 3. SIDEBAR (Chat History)
```

```
# ========================================
with st.sidebar:
    st.title("Chat History")

    if st.button("New Chat", use_container_width=True):
        new_chat_id = str(uuid.uuid4())
        st.session_state.chats[new_chat_id] = []
        st.session_state.current_chat_id = new_chat_id
        st.rerun()

    st.divider()
    st.write("**Previous Conversations:**")

    for chat_id, messages in st.session_state.chats.items():
        chat_title = messages[0]["content"][:25] + "..." if messages else "New Chat"
        if st.button(chat_title, key=chat_id, use_container_width=True):
            st.session_state.current_chat_id = chat_id
            st.rerun()

# ========================================
# 4. MAIN PAGE & AI INIT
# ========================================
st.title("Dynamic AI Data Analyst (SQLCoder 7B Agent)")
st.markdown("Upload **any** CSV file and the AI will learn its columns so you can ask questions in plain
English.")

# Try to wake the engine. If Ollama isn't running, fail loudly and stop.
try:
    with st.spinner("Waking up the SQLCoder engine..."):
        llm = load_llm()
        retriever = load_rag_memory()
        # Cheap ping so we fail fast if the server is down
        _ = llm.invoke("ping")
except Exception as e:
    st.error(
        "Could not reach the Ollama server. Is `ollama serve` running and is the "
        f"`sqlcoder:7b` model pulled?\n\nDetails: {e}"
    )
    st.stop()

# ========================================
# 5. DATABASE CONNECTION + SCHEMA
# ========================================
db_path = st.text_input("Enter the path to your DuckDB database file", value="telecom.duckdb")

if db_path and os.path.exists(db_path):
    # Open the persistent DuckDB database file directly (read-only so the
    # analyst session can never mutate the source data).
    con = duckdb.connect(db_path, read_only=True)

    # List the tables in the database and let the analyst choose one.
    tables = [row[0] for row in con.execute("SHOW TABLES").fetchall()]
    table_name = st.selectbox("Select a table to query", tables)

    # Pull the chosen table into a dataframe for preview and schema extraction.
    df = con.execute(f'SELECT * FROM "{table_name}"').df()

    def get_dynamic_schema(dataframe: pd.DataFrame, table_name: str = "user_data") -> str:
        """Build a schema string. Column names are double-quoted so spaces/special
        chars don't break the generated SQL."""
        type_mapping = {
            "int64": "INTEGER",
            "Int64": "INTEGER",
            "int32": "INTEGER",
            "float64": "FLOAT",
            "float32": "FLOAT",
            "object": "TEXT",
            "string": "TEXT",
            "bool": "BOOLEAN",
            "datetime64[ns]": "TIMESTAMP",
        }
        cols = []
        for col, dtype in dataframe.dtypes.items():
            sql_type = type_mapping.get(str(dtype), "TEXT")
```

```
            cols.append(f'"{col}" {sql_type}')
        return f'Table: {table_name}(' + ", ".join(cols) + ")"

    schema_str = get_dynamic_schema(df, table_name)

    with st.expander("View Data & Schema Extracted", expanded=False):
        st.write(f"**Extracted Schema:** `{schema_str}`")
        st.dataframe(df.head(5), use_container_width=True)

    st.markdown("---")

    # ========================================
    # 6. CONTINUOUS CHAT INTERFACE
    # ========================================
    current_messages = st.session_state.chats[st.session_state.current_chat_id]

    # Re-draw past messages
    for message in current_messages:
        with st.chat_message(message["role"]):
            st.markdown(message["content"])

            if message.get("df") is not None:
                col1, col2 = st.columns([2, 1])
                with col1:
                    st.dataframe(message["df"], use_container_width=True)
                with col2:
                    st.code(message["sql"], language="sql")
            elif message.get("error") is not None:
                st.info(f"Database Error: {message['error']}")
                st.code(message["sql"] or "-- No SQL generated", language="sql")

    # Input box
    if user_question := st.chat_input("Example: 'Show me the top 5 rows' or 'Calculate the moving
average...'"):

        with st.chat_message("user"):
            st.markdown(user_question)
        current_messages.append({"role": "user", "content": user_question})

        with st.chat_message("assistant"):
            with st.spinner("Agent is planning and writing SQL..."):

                dynamic_sql_prompt = PromptTemplate(
                    input_variables=["question", "memory_examples", "schema", "feedback"],
                    template="""You are an expert BI data analyst.
                    Database schema:
                    [table and column definitions for the active dataset]
                    Helpful Patterns:
                    [example questions and their SQL, retrieved from the knowledge base]
                    Task: [the user's natural-language question]
                    [feedback from a previous failed attempt, if any]
                    Rules:
                    1. Return only valid DuckDB SQL inside a single ```sql block.
                    2. Do not add explanations.
                    3. Use the exact column names from the schema, quoting any with special characters.
                    4. Prefer USING() over ON when joining.
                                                        """,)

                # Retrieve memory
                retrieved_docs = retriever.invoke(user_question)
                memory_string = "".join(
                            f"-- Pattern: {d.page_content}\n{d.metadata['sql']}\n\n" for d in
        retrieved_docs
                )
                if not memory_string:
                    memory_string = "No relevant examples."

                # ---- Self-healing loop ----
                max_retries = 3
                success = False
                clean_sql = ""
                error_msg = ""
                result_df = None
                attempt = 0
```

```
previous_sql = "None"
current_error = "None"

try:
    for attempt in range(1, max_retries + 1):
        if attempt > 1:
            feedback_prompt = (
                f"\nWARNING: You previously wrote this SQL:\n{previous_sql}\n\n"
                f"But it failed with this DuckDB error:\n{current_error}\n\n"
                                                    "Please FIX the SQL to resolve
                        this error. Return ONLY the fixed SQL in a ```sql block."
            )
        else:
          feedback_prompt = ""

      current_prompt = dynamic_sql_prompt.format(
          question=user_question,
          memory_examples=memory_string.strip(),
          schema=schema_str,
          feedback=feedback_prompt,
      )

      raw_output = llm.invoke(current_prompt)

      sql_match = re.search(r"```sql\s*\n(.*?)\n```", raw_output, re.DOTALL |
      re.IGNORECASE)
      if sql_match:
          clean_sql = sql_match.group(1).strip()
      else:
          # Fallback: take the first statement we can find
          stripped = raw_output.strip()
          clean_sql = stripped.split(";")[0].strip() + ";"

      try:
          result_df = con.execute(clean_sql).df()
          success = True
          break
      except Exception as e:
          current_error = str(e).splitlines()[0]
          previous_sql = clean_sql
          error_msg = current_error

except Exception as fatal_e:
    error_msg = f"Ollama Engine Error: {fatal_e}"
    success = False

# Render results
if success:
    success_msg = (
        f"**Query Executed Successfully!** "
        f"*(Solved in {attempt} attempt{'s' if attempt > 1 else ''})*"
    )
    st.markdown(success_msg)
    col1, col2 = st.columns([2, 1])
    with col1:
        st.dataframe(result_df, use_container_width=True)
    with col2:
        st.code(clean_sql, language="sql")

    current_messages.append({
        "role": "assistant",
        "content": success_msg,
        "df": result_df,
        "sql": clean_sql,
        "error": None,
    })
else:
    fail_msg = f"**Agent failed to fix the query after {max_retries} attempts.**"
    st.markdown(fail_msg)
    st.info(f"Final Error: {error_msg or 'Unknown error'}")
    st.code(clean_sql if clean_sql else "-- No SQL generated", language="sql")

    current_messages.append({
        "role": "assistant",
```

```
                "content": fail_msg,
                "df": None,
                "sql": clean_sql,
                "error": error_msg,
            })

# Force the input box to reset for the next question
st.rerun()
```

Appendix C — Evaluation Harness Code

The following listing reproduces the evaluation harness used to measure execution success and result correctness: it connects to the telecom dataset in a DuckDB database file, indexes the SQL-pattern knowledge base in FAISS, runs the SQLCoder-7B model with chain-of-thought retrieval over the question set, and scores each generated query against the reference result set.

```
import pandas as pd
import numpy as np
import duckdb
import re
from langchain_community.llms import Ollama
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_core.documents import Document

# ========================================
# 1. SETUP MODEL & EMBEDDINGS (RAG INIT)
# ========================================
print("1. Loading Defog SQLCoder 7B & RAG Embedding Model... [HARD MODE]")
llm = Ollama(
    model="sqlcoder:7b",
    temperature=0,
    num_ctx=8192,      # Expand the memory buffer so it can read the whole prompt
    num_predict=2048   # Allow it to write long answers
)

# We use a fast, lightweight sentence transformer for local embeddings
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# ========================================
# 2. BUILD THE RAG KNOWLEDGE BASE (Golden Examples)
# ========================================
print("2. Building SQL Vector Database with Advanced Patterns...")
sql_knowledge_base = [
    # Standard Patterns
    {"task": "Rank items based on a sum within a group.", "sql": "SELECT id, category, RANK()
OVER(PARTITION BY category ORDER BY sum_val DESC) FROM (SELECT id, category, SUM(val) as sum_val FROM
my_table GROUP BY 1, 2);"},
    {"task": "Show a running total or cumulative sum across time.", "sql": "SELECT id, month, SUM(val)
OVER(PARTITION BY id ORDER BY month) FROM my_table;"},
    {"task": "Calculate the difference between current row and previous row (Lag).", "sql": "SELECT id,
time, val - LAG(val) OVER(PARTITION BY id ORDER BY time) FROM my_table;"},
    {"task": "Show the value of the next row (Lead).", "sql": "SELECT id, time, LEAD(val) OVER(PARTITION BY
id ORDER BY time) FROM my_table;"},
    {"task": "Divide all items into equal groups or buckets.", "sql": "SELECT id, NTILE(10) OVER(ORDER BY
val DESC) FROM my_table;"},
    {"task": "Filter groups using HAVING after a JOIN.", "sql": "SELECT t1.category FROM table1 t1 JOIN
table2 t2 USING(id) GROUP BY t1.category HAVING COUNT(*) > 10;"},
    {"task": "Find percentage of total across all groups.", "sql": "SELECT category, SUM(val) * 100.0 /
SUM(SUM(val)) OVER() FROM my_table GROUP BY category;"},
    {"task": "Find items that exist in one condition AND another (Intersect).", "sql": "SELECT id FROM
my_table WHERE type = 'A' INTERSECT SELECT id FROM my_table WHERE type = 'B';"},
    {"task": "Calculate a 2-period moving average.", "sql": "SELECT id, time, AVG(val) OVER(PARTITION BY id
ORDER BY time ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) FROM my_table;"},
    {"task": "Filter by comparing a value to an aggregated average from another table.", "sql": "WITH
avg_cte AS (SELECT AVG(val) as avg_val FROM table1) SELECT id FROM table2 WHERE val > (SELECT avg_val FROM
avg_cte);"},
    # Advanced / Hard Mode Patterns
    {"task": "Pivot rows into columns based on a condition.", "sql": "SELECT id, SUM(CASE WHEN type = 'A'
THEN val ELSE 0 END) as A_val, SUM(CASE WHEN type = 'B' THEN val ELSE 0 END) as B_val FROM my_table GROUP
BY id;"},
    {"task": "Filter out the top and bottom percentiles or ranks.", "sql": "WITH ranked AS (SELECT id,
RANK() OVER(PARTITION BY category ORDER BY val ASC) as r_asc, RANK() OVER(PARTITION BY category ORDER BY
val DESC) as r_desc FROM my_table) SELECT id FROM ranked WHERE r_asc > 1 AND r_desc > 1;"},
```

```
    {"task": "Count items conditionally using SUM and CASE WHEN.", "sql": "SELECT id FROM my_table GROUP BY
id HAVING SUM(CASE WHEN status = 'Fail' THEN 1 ELSE 0 END) = 0;"}
]

# Convert to LangChain Documents and load into FAISS
docs = [Document(page_content=item["task"], metadata={"sql": item["sql"]}) for item in sql_knowledge_base]
vector_db = FAISS.from_documents(docs, embeddings)

# ========================================
# 3. DATABASE CONNECTION
# ========================================
print("3. Connecting to Telecom Database...")
DB_PATH = "telecom.duckdb"
con = duckdb.connect(DB_PATH, read_only=True)

# Build the schema-context string by reading each table's columns from the database.
schema_parts = []
for (table_name,) in con.execute("SHOW TABLES").fetchall():
    cols = [row[0] for row in con.execute(f'DESCRIBE "{table_name}"').fetchall()]
    schema_parts.append(f"{table_name}({', '.join(cols)})")
schema_context = "Tables: " + ", ".join(schema_parts)

# ========================================
# 4. THE 50-QUESTION GAUNTLET (20 Simple / 15 Moderate / 15 Complex)
# ========================================
exam_questions = [
    # ---------------- 20 SIMPLE ----------------
    {"q": "Count the total number of customers.",
     "exp": "SELECT COUNT(*) FROM customers;"},
    {"q": "List all distinct plan types.",
     "exp": "SELECT DISTINCT plan FROM customers;"},
    {"q": "Count the number of customers in each plan.",
     "exp": "SELECT plan, COUNT(*) FROM customers GROUP BY plan;"},
    {"q": "Find the total number of service tickets.",
     "exp": "SELECT COUNT(*) FROM service_tickets;"},
    {"q": "List all distinct priority levels in service tickets.",
     "exp": "SELECT DISTINCT priority FROM service_tickets;"},
    {"q": "Calculate the average monthly bill across all records.",
     "exp": "SELECT AVG(bill) FROM monthly_usage;"},
    {"q": "Find the maximum gb_used in a single month.",
     "exp": "SELECT MAX(gb_used) FROM monthly_usage;"},
    {"q": "Find the minimum bill recorded.",
     "exp": "SELECT MIN(bill) FROM monthly_usage;"},
    {"q": "Count the number of service tickets per priority level.",
     "exp": "SELECT priority, COUNT(*) FROM service_tickets GROUP BY priority;"},
    {"q": "List the cust_id and bill for all usage records in Month 1.",
     "exp": "SELECT cust_id, bill FROM monthly_usage WHERE month_num = 1;"},
    {"q": "Find all customers on the 'Enterprise' plan.",
     "exp": "SELECT cust_id FROM customers WHERE plan = 'Enterprise';"},
    {"q": "Calculate the total bill summed across every record.",
     "exp": "SELECT SUM(bill) FROM monthly_usage;"},
    {"q": "Count how many 'High' priority tickets exist.",
     "exp": "SELECT COUNT(*) FROM service_tickets WHERE priority = 'High';"},
    {"q": "Find the average gb_used for Month 2.",
     "exp": "SELECT AVG(gb_used) FROM monthly_usage WHERE month_num = 2;"},
    {"q": "List usage records where the bill exceeded 100.",
     "exp": "SELECT cust_id, month_num, bill FROM monthly_usage WHERE bill > 100;"},
    {"q": "Show the total bill for each customer across all months.",
     "exp": "SELECT cust_id, SUM(bill) FROM monthly_usage GROUP BY cust_id;"},
    {"q": "Count the number of distinct customers who have at least one ticket.",
     "exp": "SELECT COUNT(DISTINCT cust_id) FROM service_tickets;"},
    {"q": "Find the average bill for each month_num.",
     "exp": "SELECT month_num, AVG(bill) FROM monthly_usage GROUP BY month_num;"},
    {"q": "List the 10 highest single monthly bills.",
     "exp": "SELECT cust_id, month_num, bill FROM monthly_usage ORDER BY bill DESC LIMIT 10;"},
    {"q": "Find the total gb_used for each customer.",
     "exp": "SELECT cust_id, SUM(gb_used) FROM monthly_usage GROUP BY cust_id;"},
    # ---------------- 15 MODERATE ----------------
    {"q": "Count the number of tickets for each plan.",
     "exp": "SELECT plan, COUNT(ticket_id) FROM customers JOIN service_tickets USING(cust_id) GROUP BY
plan;"},
    {"q": "Find the average bill per plan.",
     "exp": "SELECT plan, AVG(bill) FROM customers JOIN monthly_usage USING(cust_id) GROUP BY plan;"},
    {"q": "List customers whose total bill exceeds 300.",
```

```
     "exp": "SELECT cust_id FROM monthly_usage GROUP BY cust_id HAVING SUM(bill) > 300;"},
    {"q": "Find customers who have submitted more than 2 tickets.",
     "exp": "SELECT cust_id FROM service_tickets GROUP BY cust_id HAVING COUNT(*) > 2;"},
    {"q": "Show the plans whose average monthly gb_used is greater than 50.",
     "exp": "SELECT plan, AVG(gb_used) FROM customers JOIN monthly_usage USING(cust_id) GROUP BY plan
HAVING AVG(gb_used) > 50;"},
    {"q": "Find customers whose Month 3 gb_used was above the overall average Month 3 gb_used.",
     "exp": "SELECT cust_id FROM monthly_usage WHERE month_num = 3 AND gb_used > (SELECT AVG(gb_used) FROM
monthly_usage WHERE month_num = 3);"},
    {"q": "List the customer IDs whose total gb_used across all three months exceeds 200.",
     "exp": "SELECT cust_id FROM monthly_usage GROUP BY cust_id HAVING SUM(gb_used) > 200;"},
    {"q": "Find customers who have submitted at least one ticket of every priority level.",
     "exp": "SELECT cust_id FROM service_tickets GROUP BY cust_id HAVING COUNT(DISTINCT priority) = 3;"},
    {"q": "For each customer who opened a ticket, find the ticket_id of their first (lowest) ticket.",
     "exp": "SELECT cust_id, MIN(ticket_id) FROM service_tickets GROUP BY cust_id;"},
    {"q": "Find customers whose highest monthly bill is more than double their lowest monthly bill.",
     "exp": "SELECT cust_id FROM monthly_usage GROUP BY cust_id HAVING MAX(bill) > 2 * MIN(bill);"},
    {"q": "Count the number of 'High' priority tickets per plan.",
     "exp": "SELECT plan, COUNT(*) FROM customers JOIN service_tickets USING(cust_id) WHERE priority =
'High' GROUP BY plan;"},
    {"q": "Find the month_num that has the highest average bill across all customers.",
     "exp": "SELECT month_num, AVG(bill) FROM monthly_usage GROUP BY month_num ORDER BY AVG(bill) DESC
LIMIT 1;"},
    {"q": "List customers on the 'Premium' plan whose total gb_used exceeds 150.",
     "exp": "SELECT customers.cust_id FROM customers JOIN monthly_usage USING(cust_id) WHERE customers.plan
= 'Premium' GROUP BY customers.cust_id HAVING SUM(monthly_usage.gb_used) > 150;"},
    {"q": "Find the median monthly bill for each plan.",
     "exp": "SELECT plan, PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY bill) FROM customers JOIN
monthly_usage USING(cust_id) GROUP BY plan;"},
    {"q": "Count, for each plan, the number of customers who have never opened a ticket.",
     "exp": "SELECT customers.plan, COUNT(*) FROM customers LEFT JOIN service_tickets USING(cust_id) WHERE
service_tickets.ticket_id IS NULL GROUP BY customers.plan;"},
    # ---------------- 15 COMPLEX ----------------
    {"q": "For each customer, find the exact month number where they had their highest bill.",
     "exp": "WITH ranked AS (SELECT cust_id, month_num, RANK() OVER(PARTITION BY cust_id ORDER BY bill
DESC) as rnk FROM monthly_usage) SELECT cust_id, month_num FROM ranked WHERE rnk = 1;"},
    {"q": "Find customers whose bill strictly increased every single month (Month 1 < Month 2 < Month 3).",
     "exp": "WITH m AS (SELECT cust_id, month_num, bill FROM monthly_usage) SELECT m1.cust_id FROM m m1
JOIN m m2 ON m1.cust_id = m2.cust_id AND m2.month_num = 2 JOIN m m3 ON m1.cust_id = m3.cust_id AND
m3.month_num = 3 WHERE m1.month_num = 1 AND m1.bill < m2.bill AND m2.bill < m3.bill;"},
    {"q": "For each priority level, find the customer ID who spent the most total money across all
months.",
     "exp": "WITH cust_totals AS (SELECT cust_id, SUM(bill) as total_bill FROM monthly_usage GROUP BY
cust_id), ticket_spenders AS (SELECT t.priority, t.cust_id, c.total_bill, RANK() OVER(PARTITION BY
t.priority ORDER BY c.total_bill DESC) as rnk FROM service_tickets t JOIN cust_totals c USING(cust_id))
SELECT priority, cust_id FROM ticket_spenders WHERE rnk = 1;"},
    {"q": "Calculate the ratio of each customer's monthly bill to their own maximum monthly bill across all
months.",
     "exp": "SELECT cust_id, month_num, bill / MAX(bill) OVER(PARTITION BY cust_id) FROM monthly_usage;"},
    {"q": "Find customers on the 'Enterprise' plan who have NEVER submitted a 'High' priority ticket but
have submitted at least 2 'Low' priority tickets.",
     "exp": "SELECT c.cust_id FROM customers c JOIN service_tickets t USING(cust_id) WHERE c.plan =
'Enterprise' GROUP BY c.cust_id HAVING SUM(CASE WHEN t.priority = 'High' THEN 1 ELSE 0 END) = 0 AND
SUM(CASE WHEN t.priority = 'Low' THEN 1 ELSE 0 END) >= 2;"},
    {"q": "Which customers are in the top 5 percent of total data usage (gb_used) across the entire
dataset?",
     "exp": "WITH totals AS (SELECT cust_id, SUM(gb_used) as total_gb FROM monthly_usage GROUP BY cust_id),
percentiles AS (SELECT cust_id, PERCENT_RANK() OVER(ORDER BY total_gb DESC) as pr FROM totals) SELECT
cust_id FROM percentiles WHERE pr <= 0.05;"},
    {"q": "Show the customer IDs whose Month 2 GB usage dropped by more than 50 percent compared to their
Month 1 GB usage.",
     "exp": "WITH m1 AS (SELECT cust_id, gb_used FROM monthly_usage WHERE month_num = 1), m2 AS (SELECT
cust_id, gb_used FROM monthly_usage WHERE month_num = 2) SELECT m2.cust_id FROM m2 JOIN m1 USING(cust_id)
WHERE m2.gb_used < (m1.gb_used * 0.5);"},
    {"q": "Pivot the data to show each customer's bill for month 1, month 2, and month 3 as three separate
columns.",
     "exp": "SELECT cust_id, SUM(CASE WHEN month_num = 1 THEN bill ELSE 0 END) as m1, SUM(CASE WHEN
month_num = 2 THEN bill ELSE 0 END) as m2, SUM(CASE WHEN month_num = 3 THEN bill ELSE 0 END) as m3 FROM
monthly_usage GROUP BY cust_id;"},
    {"q": "List the customer IDs that have a higher-than-average total bill AND a higher-than-average total
number of tickets.",
     "exp": "WITH totals AS (SELECT cust_id, SUM(bill) as total_bill FROM monthly_usage GROUP BY cust_id),
tickets AS (SELECT cust_id, COUNT(*) as ticket_count FROM service_tickets GROUP BY cust_id) SELECT
t.cust_id FROM totals t JOIN tickets tk USING(cust_id) WHERE t.total_bill > (SELECT AVG(total_bill) FROM
```

```
totals) AND tk.ticket_count > (SELECT AVG(ticket_count) FROM tickets);"},
    {"q": "Find the customer who had the largest single month-over-month increase in gb_used, returning the
cust_id and the increase amount.",
     "exp": "WITH diffs AS (SELECT cust_id, gb_used - LAG(gb_used) OVER(PARTITION BY cust_id ORDER BY
month_num) as increase FROM monthly_usage) SELECT cust_id, increase FROM diffs WHERE increase IS NOT NULL
ORDER BY increase DESC LIMIT 1;"},
    {"q": "Calculate the average bill per plan, but exclude the absolute highest and lowest single bills
for each plan.",
     "exp": "WITH ranked AS (SELECT plan, bill, RANK() OVER(PARTITION BY plan ORDER BY bill ASC) as r_asc,
RANK() OVER(PARTITION BY plan ORDER BY bill DESC) as r_desc FROM customers JOIN monthly_usage
USING(cust_id)) SELECT plan, AVG(bill) FROM ranked WHERE r_asc > 1 AND r_desc > 1 GROUP BY plan;"},
    {"q": "Show the running cumulative total of ONLY 'High' priority tickets submitted, ordered by
ticket_id.",
     "exp": "SELECT ticket_id, SUM(CASE WHEN priority = 'High' THEN 1 ELSE 0 END) OVER(ORDER BY ticket_id)
FROM service_tickets;"},
    {"q": "For each customer, calculate the total number of months their bill was higher than their own
overall average monthly bill.",
     "exp": "WITH avg_bills AS (SELECT cust_id, AVG(bill) as avg_bill FROM monthly_usage GROUP BY cust_id)
SELECT u.cust_id, COUNT(u.month_num) FROM monthly_usage u JOIN avg_bills a USING(cust_id) WHERE u.bill >
a.avg_bill GROUP BY u.cust_id;"},
    {"q": "Find the plan that has the highest percentage of customers who have NEVER opened a service
ticket.",
     "exp": "WITH cust_tickets AS (SELECT c.plan, c.cust_id, CASE WHEN t.ticket_id IS NULL THEN 1 ELSE 0
END as no_ticket FROM customers c LEFT JOIN service_tickets t USING(cust_id)) SELECT plan FROM cust_tickets
GROUP BY plan ORDER BY SUM(no_ticket) * 1.0 / COUNT(DISTINCT cust_id) DESC LIMIT 1;"},
    {"q": "Find customers who had usage in Month 1 and Month 3, but NO usage at all in Month 2.",
     "exp": "SELECT cust_id FROM monthly_usage WHERE month_num = 1 INTERSECT SELECT cust_id FROM
monthly_usage WHERE month_num = 3 EXCEPT SELECT cust_id FROM monthly_usage WHERE month_num = 2;"}
]

# ========================================
# 5. RAG EXECUTION LOOP (Strict Mode + Chain of Thought)
# ========================================
print("\n=== STARTING 'HARD MODE' RAG + CoT BENCHMARK ===")
prec_score = 0
exec_score = 0

for i, test in enumerate(exam_questions, 1):
    print(f"\nTest {i}: {test['q'][:50]}...")

    # --- RAG RETRIEVAL STEP ---
    retrieved_docs = vector_db.similarity_search(test['q'], k=3)
    rag_context = "\n".join([f"-- Pattern: {d.page_content}\n{d.metadata['sql']}" for d in retrieved_docs])

    # --- CHAIN OF THOUGHT PROMPT ---
    prompt = f"""You are an expert BI data analyst.
Database schema:
[table and column definitions for the active dataset]
Helpful Patterns:
[example questions and their SQL, retrieved from the knowledge base]
Task: [the user's natural-language question]
[feedback from a previous failed attempt, if any]
Rules:
1. Return only valid DuckDB SQL inside a single ```sql block.
2. Do not add explanations.
3. Use the exact column names from the schema, quoting any with special characters.
4. Prefer USING() over ON when joining."""

    raw_response = llm.invoke(prompt)

    # Extract the AI's "Thinking" text (everything before the SQL block)
    thinking_text = raw_response.split("```")[0].strip()
    # Print the first 250 characters of its thought process so we can watch it reason!
    print(f"  AI Thinking: {thinking_text[:250]}...\n")

    # Extract SQL
    sql_match = re.search(r"```sql\n(.*?)\n```", raw_response, re.DOTALL | re.IGNORECASE)
    ai_sql = sql_match.group(1).strip() if sql_match else raw_response.strip().split(';')[0] + ';'

    try:
        df_ai = con.execute(ai_sql).df()
        exec_score += 1
        df_exp = con.execute(test['exp']).df()
```

```
        if df_ai.shape != df_exp.shape:
            print(f"  LOGIC FAIL (Shape mismatch. AI returned {df_ai.shape[1]} cols, Expected
{df_exp.shape[1]} cols)")
            print(f"     [AI SQL]: {ai_sql}")
            continue

        # Normalize and sort DataFrames for exact comparison
        df_ai.columns = [f"c{j}" for j in range(len(df_ai.columns))]
        df_exp.columns = [f"c{j}" for j in range(len(df_exp.columns))]
        df_ai = df_ai.sort_values(by=list(df_ai.columns)).reset_index(drop=True)
        df_exp = df_exp.sort_values(by=list(df_exp.columns)).reset_index(drop=True)

        pd.testing.assert_frame_equal(df_ai, df_exp, check_dtype=False, atol=1e-2)
        print("  PASS")
        prec_score += 1

    except AssertionError:
        print("  LOGIC FAIL (Values don't match)")
        print(f"     [AI SQL]: {ai_sql}")
    except Exception as e:
        print(f"  CRASH: {str(e).splitlines()[0]}")
        print(f"     [AI SQL]: {ai_sql}")

print("\n" + "=" * 40)
print("FINAL RAG RESULTS 'HARD MODE' (Defog SQLCoder 7B + CoT):")
print(f"Execution: {(exec_score / len(exam_questions)) * 100:.1f}%")
print(f"Precision: {(prec_score / len(exam_questions)) * 100:.1f}%")
print("=" * 40)
```

Appendix D — Hardware Specifications

Development and testing tier (single workstation): quad-core Intel Core i7 or equivalent CPU; 32 GB RAM; NVIDIA RTX 3060 or equivalent GPU for local inference; 1 TB SSD; stable broadband connection for software updates and integration testing.
Production tier (server): server-class CPU (Intel Xeon or equivalent); 16 to 32 GB RAM; NVIDIA RTX 3060-class or higher GPU for inference acceleration; 512 GB to 1 TB SSD; high-throughput network connection for remote analyst access. Production deployment in a multi-user environment would require horizontal scaling beyond a single server, the design of which is out of scope for this thesis.

Appendix E — Evaluation Query Set

The complete set of fifty natural-language questions used during system development and internal evaluation is listed in Table E.1 below. Each entry gives the question, its difficulty band, and the key SQL features it exercises. Three representative examples are reproduced in the main text (Section 4.6.1).

#	Difficulty	Natural-language question	Key SQL features
1	Simple	Count the total number of customers.	Single table; COUNT
2	Simple	List all distinct plan types.	Single table; DISTINCT
3	Simple	Count the number of customers in each plan.	Single table; GROUP BY, COUNT
4	Simple	Find the total number of service tickets.	Single table; COUNT
5	Simple	List all distinct priority levels in service tickets.	Single table; DISTINCT
6	Simple	Calculate the average monthly bill across all records.	Single table; AVG
7	Simple	Find the maximum gb_used in a single month.	Single table; MAX
8	Simple	Find the minimum bill recorded.	Single table; MIN
9	Simple	Count the number of service tickets per priority level.	Single table; GROUP BY, COUNT
10	Simple	List the cust_id and bill for all usage records in Month 1.	Single table; WHERE filter
11	Simple	Find all customers	Single table; WHERE

#	Difficulty	Natural-language question	Key SQL features
		on the 'Enterprise' plan.	filter
12	Simple	Calculate the total bill summed across every record.	Single table; SUM
13	Simple	Count how many 'High' priority tickets exist.	Single table; COUNT with filter
14	Simple	Find the average gb_used for Month 2.	Single table; AVG with filter
15	Simple	List usage records where the bill exceeded 100.	Single table; WHERE filter
16	Simple	Show the total bill for each customer across all months.	Single table; GROUP BY, SUM
17	Simple	Count distinct customers who have at least one ticket.	Single table; COUNT DISTINCT
18	Simple	Find the average bill for each month_num.	Single table; GROUP BY, AVG
19	Simple	List the 10 highest single monthly bills.	Single table; ORDER BY, LIMIT
20	Simple	Find the total gb_used for each customer.	Single table; GROUP BY, SUM
21	Moderate	Count the number of tickets for each plan.	Join customers–tickets; GROUP BY, COUNT
22	Moderate	Find the average bill per plan.	Join customers–usage; GROUP BY, AVG
23	Moderate	List customers whose total bill exceeds 300.	GROUP BY, SUM, HAVING
24	Moderate	Find customers who have submitted more than 2 tickets.	GROUP BY, COUNT, HAVING
25	Moderate	Show the plans whose average monthly gb_used is greater than 50.	Join; GROUP BY, HAVING
26	Moderate	Find customers whose Month-3 gb_used was above the overall Month-3 average.	Subquery; WHERE with aggregate
27	Moderate	List customer IDs	GROUP BY, SUM,

#	Difficulty	Natural-language question	Key SQL features
		whose total gb_used across all months exceeds 200.	HAVING
28	Moderate	Find customers who have submitted a ticket of every priority level.	GROUP BY, COUNT DISTINCT, HAVING
29	Moderate	For each customer with a ticket, find their first (lowest) ticket_id.	GROUP BY, MIN
30	Moderate	Find customers whose highest monthly bill is more than double their lowest.	GROUP BY, MAX/MIN, HAVING
31	Moderate	Count the number of 'High' priority tickets per plan.	Join; GROUP BY, COUNT with filter
32	Moderate	Find the month_num with the highest average bill across all customers.	GROUP BY, AVG, ORDER BY, LIMIT
33	Moderate	List 'Premium' customers whose total gb_used exceeds 150.	Join; filter; GROUP BY, HAVING
34	Moderate	Find the median monthly bill for each plan.	Join; GROUP BY; PERCENTILE_CONT
35	Moderate	For each plan, count customers who have never opened a ticket.	Anti-join (LEFT JOIN … IS NULL); GROUP BY
36	Complex	For each customer, find the month with their highest bill.	Window: RANK over partition
37	Complex	Find customers whose bill increased every month (M1<M2<M3).	Self-join across periods
38	Complex	For each priority level, find the customer who spent the most overall.	CTE + RANK over partition
39	Complex	Ratio of each customer's monthly	Window: MAX OVER (PARTITION BY)

#	Difficulty	Natural-language question	Key SQL features
		bill to their own maximum bill.	
40	Complex	'Enterprise' customers with no 'High' ticket but ≥2 'Low' tickets.	Join; conditional SUM/CASE; HAVING
41	Complex	Identify customers in the top 5% of total data usage.	CTE; PERCENT_RANK window
42	Complex	Customers whose Month-2 GB dropped >50% versus Month 1.	Self-join across periods
43	Complex	Pivot each customer's bill for months 1–3 into three columns.	Conditional aggregation / pivot
44	Complex	Customers with above-average total bill AND above-average ticket count.	Multiple CTEs; subquery averages
45	Complex	Customer with the largest month-over-month gb_used increase.	LAG; ordering
46	Complex	Average bill per plan, excluding each plan's highest and lowest bill.	Window RANK both directions; filter
47	Complex	Running cumulative total of only 'High' priority tickets, by ticket_id.	Window: cumulative conditional SUM
48	Complex	For each customer, count months their bill exceeded their own average.	CTE average; conditional count
49	Complex	Plan with the highest percentage of customers who never opened a ticket.	LEFT JOIN; ratio; ORDER BY, LIMIT
50	Complex	Customers with usage in Month 1 and Month 3 but none in Month 2.	INTERSECT / EXCEPT set operations

Table E.1 — The complete fifty-question evaluation set used during system development and internal evaluation (see Section 4.6.1).