Got it. I’ll analyze the LLaMA Code 7B Instruct model’s architecture, training data, prompting strategies, and published best practices to create a detailed guide for optimizing natural language to Postgres SQL translation using this model.

I’ll include:

* An optimal prompt tailored to your schema and task
* Ideal model parameters for accuracy and consistency
* Tips to improve join handling, column selection, and SQL correctness.

I’ll get back to you shortly with the complete report.

**Optimizing NL-to-SQL Query Generation with LLaMA Code 7B Instruct**

**1. Optimum Prompt Design for NL-to-SQL Conversion**

Designing a clear, context-rich prompt is crucial when using LLaMA Code 7B Instruct to translate natural language to PostgreSQL SQL. The prompt should explicitly provide the model with the database schema and instructions on how to form the query. Key elements include:

* **Include the Schema:** Provide table names, columns, and relationships so the model knows what it can query ([Prompt Engineering for a Better SQL Code Generation With LLMs | by Raghid Bsat | Dataminded | Medium](https://medium.com/datamindedbe/prompt-engineering-for-a-better-sql-code-generation-with-llms-263562c0c35d#:~:text=1)). For example, present the schema as part of the prompt (or system message) in a structured format. This could be a short description or even a DDL snippet of each table. Including the schema gives necessary context for the model to pick correct tables/columns ([Exploring Different Capabilities of CodeLlama](https://www.analyticsvidhya.com/blog/2024/07/different-capabilities-of-codellama/#:~:text=Below%20is%20the%20example%20where,SQL%20query%20with%20database%20schema)) ([Exploring Different Capabilities of CodeLlama](https://www.analyticsvidhya.com/blog/2024/07/different-capabilities-of-codellama/#:~:text=table1%3D%20,10)). In our case, we might include something like:

*Schema:*  
branches(branch\_id, branch\_name, governorate, branch\_address, latitude, longitude)  
transactions(transaction\_id, transaction\_currency, transaction\_type, transaction\_amount, branch\_id)

This ensures the model is aware of the two tables and the linking key (branch\_id). Studies have shown that giving an LLM the database schema up front significantly improves accuracy of the generated SQL ([Prompt Engineering for a Better SQL Code Generation With LLMs | by Raghid Bsat | Dataminded | Medium](https://medium.com/datamindedbe/prompt-engineering-for-a-better-sql-code-generation-with-llms-263562c0c35d#:~:text=1)).

* **Specify the SQL Dialect (PostgreSQL):** Make it clear the query should be in PostgreSQL syntax. LLMs trained on code can produce MySQL-like or generic SQL by default ([Natural Language Queries to SQL with Code-Llama | E2E Cloud](https://docs.e2enetworks.com/docs/tir/Inference/Tutorials/Natural_Language_Queries_to_SQL_with_Code_Llama/#:~:text=1)). Stating "Write a PostgreSQL query..." in the prompt helps avoid dialect issues (e.g. use of CURRENT\_DATE or NOW() for current date instead of MySQL’s DATE\_SUB syntax). Tailoring the prompt to the specific SQL dialect prevents syntax errors due to dialect differences ([Natural Language Queries to SQL with Code-Llama | E2E Cloud](https://docs.e2enetworks.com/docs/tir/Inference/Tutorials/Natural_Language_Queries_to_SQL_with_Code_Llama/#:~:text=1)).
* **Explicitly Instruct on JOINs:** The prompt should remind the model to use proper join conditions when columns from multiple tables are needed. For example: *"Use an INNER JOIN between branches and transactions on matching branch\_id to combine data from both tables."* Emphasize that to retrieve columns across tables, a join on branch\_id is required (since branch\_id is the foreign key linking the two tables). This directly addresses the model’s common mistake of selecting from multiple tables without a join. You can even include a brief note in the prompt like: *"Remember: combine tables with ... JOIN branches b ON b.branch\_id = transactions.branch\_id when needed."* Providing such guidance in the prompt has been found to improve the model’s handling of multi-table queries ([NSQL-Llama-2-7b: Advancing SQL Foundation Models for Personalized Analytics - Numbers Station](https://www.numbersstation.ai/nsql-llama-2-7b/#:~:text=we%20show%20that%20NSQL,%E2%80%8D)) (models with explicit training in SQL show marked improvement in choosing the correct join strategy).
* **Quote Identifiers When Needed:** Instruct the model to double-quote identifiers if they are reserved words or contain spaces/special characters. For example: *"If a column or table name has spaces or special characters, use double quotes (e.g., \"Transaction Type\")."* In our given schema, all identifiers are lowercase and underscore-separated (which are safe unquoted), but this prompt instruction is a safeguard for other schemas. It prevents syntax errors in cases where quoting is necessary. The model, being a code-focused LLM, will typically include quotes if the prompt encourages it and if it sees unusual characters in names ([Natural Language Queries to SQL with Code-Llama | E2E Cloud](https://docs.e2enetworks.com/docs/tir/Inference/Tutorials/Natural_Language_Queries_to_SQL_with_Code_Llama/#:~:text=It%20should%20be%20ensured%20that,it%E2%80%99s%20output%20is%20given%20below)).
* **Encourage LIMIT for Large Results:** To make the query output manageable and efficient, the prompt can suggest adding a LIMIT clause. For instance: *"If the query might return many rows, add LIMIT 10."* This is especially useful in an interactive setting where we don't want the model to generate a huge result set. By including this in the instructions, the model will learn to append something like LIMIT 5 or LIMIT 10 to the SQL for broad queries (e.g., “List all transactions for branch X”) to avoid overwhelming outputs. This practice has been noted in prompt-engineering guides to help keep results concise ([Natural Language Queries to SQL with Code-Llama | E2E Cloud](https://docs.e2enetworks.com/docs/tir/Inference/Tutorials/Natural_Language_Queries_to_SQL_with_Code_Llama/#:~:text=It%20should%20be%20ensured%20that,it%E2%80%99s%20output%20is%20given%20below)).
* **Use of Dates and Now():** If the user’s question implies a current date/time filter (e.g., “... as of today” or “this month”), instruct the model to use PostgreSQL date functions. For example: *"Use CURRENT\_DATE or DATE(NOW()) for 'today's date' in conditions."* This ensures the model knows to incorporate the current date dynamically. For instance, *“…WHERE transaction\_date >= DATE(NOW()) - INTERVAL '30 days'”* for “in the last month” (appropriate Postgres syntax), or simply = CURRENT\_DATE for “today”. Being explicit in the prompt about using date('now') or CURRENT\_DATE helps the model choose the correct function rather than leaving it to guess.

**Prompt Structure:** LLaMA Code 7B Instruct follows the LLaMA-2 style instruction format ([codellama/CodeLlama-7b-Instruct-hf · code llama prompt template](https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf/discussions/19" \l ":~:text=The%20conversational%20instructions%20follow%20the,of%20the%20delimiters%20for%20you)). In practice, you might format the prompt with a system message that contains the guidelines and schema, and then the user query. For example, using the HuggingFace chat template (which inserts special tokens like [INST]):

**System (Instruction)**: *"You are an expert SQL generator. Given a database schema and a user's request in natural language, your job is to output a syntactically correct PostgreSQL SQL query that answers the question. Use the schema to identify the correct tables/columns. Ensure to join tables on matching keys when needed (for example, use branches.branch\_id = transactions.branch\_id to join the two tables). Quote identifiers with double quotes if they contain spaces or are reserved words. If a query could return many rows, include LIMIT 10. Use CURRENT\_DATE (PostgreSQL) for 'today' if needed. Do not explain the query in words, only provide the SQL statement."*

**User (Query)**: *"Which branches have a total transaction amount over 10000 USD this month?"*

Such a prompt provides context and clear instructions. In a programmatic setting, this can be constructed as a single string or using the chat format. For example, one open-source implementation of text-to-SQL with Code Llama 7B creates a system prompt with the schema and instructions, then appends the user question, using the official chat template to combine them ([finetune-LLMs/app.py at main · Pyligent/finetune-LLMs · GitHub](https://github.com/Pyligent/finetune-LLM/blob/main/app.py#:~:text=schema%20%3D%20%27You%20are%20an,school%20VARCHAR%2C%20last_occ_championship%20VARCHAR)) ([finetune-LLMs/app.py at main · Pyligent/finetune-LLMs · GitHub](https://github.com/Pyligent/finetune-LLM/blob/main/app.py#:~:text=combined_json_data%20%3D%20)). By crafting the prompt this way, you set the model up for success: it knows the relevant tables/columns, the required join key, and special considerations (quoting, limits, date functions) before it starts generating SQL.

**2. Recommended Model Parameters for SQL Generation**

Choosing the right generation parameters can significantly improve the quality and correctness of SQL outputs. For LLaMA Code 7B Instruct, which is a code-focused model, the goal is to make the output as deterministic and syntactically correct as possible. Here are the recommended parameters and settings:

* **Decoding Strategy:** Use a *deterministic or low-temperature decoding*. For production SQL generation, consistency is more important than creativity. A common approach is to turn off sampling (do\_sample=False) to use greedy decoding ([finetune-LLMs/app.py at main · Pyligent/finetune-LLMs · GitHub](https://github.com/Pyligent/finetune-LLM/blob/main/app.py#:~:text=prompt%20%3D%20pipe)). Greedy or nearly-greedy decoding will pick the highest probability next token each time, which often yields a well-structured query. If you do use sampling, set a very low temperature (e.g. 0.1 or 0.2) ([codellama/CodeLlama-34b-hf · Hugging Face](https://huggingface.co/codellama/CodeLlama-34b-hf" \l ":~:text=sequences%20%3D%20pipeline%28%20%27import%20socket,eos_token_id%2C%20max_length%3D200)) to reduce randomness. Low temperature ensures the model doesn't stray into syntactically incorrect or irrelevant output due to sampling variability.
* **Top-p and Top-k:** If sampling is enabled, constrain it heavily. For example, top\_p=0.9 or 0.95 and top\_k=50 are reasonable defaults that allow some flexibility but filter out low-probability tokens ([codellama/CodeLlama-34b-hf · Hugging Face](https://huggingface.co/codellama/CodeLlama-34b-hf" \l ":~:text=top_k%3D10%2C%20temperature%3D0,Result%3A%20%7Bseq%5B%27generated_text)). In some cases, an even more restrictive setting like top\_p=0.1 with top\_k=50 has been used for text-to-SQL tasks ([finetune-LLMs/app.py at main · Pyligent/finetune-LLMs · GitHub](https://github.com/Pyligent/finetune-LLM/blob/main/app.py#:~:text=prompt%20%3D%20pipe)) (meaning the model only considers the top 10% probability mass for next tokens, and among the top 50 options). Such strict settings can make the output more predictable. In summary: keep top\_p relatively high (to include the main probable tokens) but not 1.0, and top\_k moderate; or simply rely on greedy decoding which inherently picks the most likely token each time.
* **Max New Tokens / Max Length:** Ensure the model has enough budget to output the entire SQL query. Complex queries with joins and subqueries can be a few hundred tokens. Setting max\_new\_tokens (or max\_length for the total prompt+output) to a value like 256 or 512 is usually plenty for a single SQL query. For simpler queries, the model will naturally stop earlier (especially if it outputs a closing ; or just stops at a logical end). In practice, 200-256 tokens is often sufficient for one query ([Exploring Different Capabilities of CodeLlama](https://www.analyticsvidhya.com/blog/2024/07/different-capabilities-of-codellama/#:~:text=prompt%20%3D%20f,question%7D%3A%20%60%60%60)), but allocate more if you expect very elaborate SQL.
* **Stop Sequences:** Define stop criteria to prevent the model from rambling beyond the SQL. Often the model will stop on its own after producing a valid query (especially if using an eos\_token\_id). It’s good to set the eos\_token\_id to the model’s end-of-sequence token so that generation will halt if the model decides the answer is complete ([codellama/CodeLlama-34b-hf · Hugging Face](https://huggingface.co/codellama/CodeLlama-34b-hf" \l ":~:text=top_k%3D10%2C%20temperature%3D0,Result%3A%20%7Bseq%5B%27generated_text)). Additional stop sequences can be employed if needed. For instance, you could stop on a newline-newline pattern or a semicolon (;) followed by a newline, under the assumption that the query will end with ;. Research experiments have used stop tokens like double newlines or comment tokens to cut off any extra explanation ([[2204.00498] Evaluating the Text-to-SQL Capabilities of Large Language Models](https://ar5iv.org/abs/2204.00498#:~:text=A)). In a production system, you might simply post-process to trim any text after a ;. If you instruct the model to output *only* the SQL, it often won’t add extraneous text, but a safety stop (like looking for ; or the </s> token) can be added.
* **Repetition Penalty:** Apply a slight repetition penalty (e.g. 1.1 or 1.2) if you observe the model tends to repeat phrases or list unnecessary columns. Repetition penalty penalizes the model for generating the same token again and again, which can prevent it from echoing parts of the question or repeating a FROM clause twice. While Code Llama models are less prone to looping than some older models, a small repetition penalty (around 1.1) is a good safeguard to ensure the model doesn’t, for example, keep generating JOIN ... JOIN ... incorrectly. This parameter often isn’t mandatory, but it can improve reliability for edge cases.
* **Batch and Parallel Settings:** (Optional) In a production setup, you might generate multiple queries or handle multiple users. The LLaMA 7B model can handle batch generation if the library supports it. This doesn’t affect single-query quality but helps throughput. Just ensure each prompt fits in context (7B models typically handle 4k tokens context; Code Llama supports up to 100k in some cases ([Exploring Different Capabilities of CodeLlama](https://www.analyticsvidhya.com/blog/2024/07/different-capabilities-of-codellama/#:~:text=input%20contexts%2C%20and%20zero,on%20HumanEval%20and%20MBPP%2C%20respectively)), though using extremely large context may slow inference).

Putting it together, an example configuration using HuggingFace Transformers pipeline might look like:

output = pipeline(prompt,

do\_sample=False,

temperature=0.1,

top\_p=0.9, top\_k=50,

max\_new\_tokens=256,

repetition\_penalty=1.1,

eos\_token\_id=tokenizer.eos\_token\_id)

This configuration will make the model very focused and deterministic in its output ([finetune-LLMs/app.py at main · Pyligent/finetune-LLMs · GitHub](https://github.com/Pyligent/finetune-LLM/blob/main/app.py#:~:text=prompt%20%3D%20pipe)). In one open implementation of a CodeLlama 7B text-to-SQL generator, for instance, they use greedy decoding (do\_sample=False) with temperature=0.1, top\_k=50, top\_p=0.1 and max\_new\_tokens=256 ([finetune-LLMs/app.py at main · Pyligent/finetune-LLMs · GitHub](https://github.com/Pyligent/finetune-LLM/blob/main/app.py#:~:text=prompt%20%3D%20pipe)). These settings yielded accurate SQL queries by avoiding randomness. You can adjust slighty (e.g. temperature up to 0.2 or top\_p to 0.95) if you find the model is too rigid or truncating valid alternatives, but generally a low temperature and limited sampling space works best for SQL generation.

Finally, remember to include the model's end-of-sequence token and a padding token if needed (the HuggingFace pipeline example above sets eos\_token\_id and will use the model’s default pad token) to properly terminate the generation ([codellama/CodeLlama-34b-hf · Hugging Face](https://huggingface.co/codellama/CodeLlama-34b-hf" \l ":~:text=top_k%3D10%2C%20temperature%3D0,Result%3A%20%7Bseq%5B%27generated_text)). With these parameters, the model is configured to produce concise, correct SQL most of the time.

**3. Techniques to Improve SQL Accuracy (Joins, Disambiguation, and Syntax)**

Even with a good prompt and decoding settings, generating perfect SQL can be challenging. LLaMA Code 7B, as a general code model, may occasionally misinterpret the query or schema. Here are several tips and tricks to boost accuracy and reliability, drawn from best practices and insights in the literature:

* **Provide Schema Context and Relationships:** As emphasized, always give the model the schema and the foreign key relationships in the prompt. Ambiguities in the user’s request can often be resolved if the model knows how tables relate. For example, if the user asks “Show me the total transactions per branch,” the model needs to know that transactions has a branch\_id linking to branches. By including a note like "*branches.branch\_id is the primary key referenced by transactions.branch\_id*" in the context, you guide the model to use the correct join. Lack of schema context is a common source of errors in NL-to-SQL ([Prompt Engineering for a Better SQL Code Generation With LLMs | by Raghid Bsat | Dataminded | Medium](https://medium.com/datamindedbe/prompt-engineering-for-a-better-sql-code-generation-with-llms-263562c0c35d#:~:text=In%20the%20case%20of%20complex,main%20issues%20the%20LLM%20faced)). In practice, this might mean prepending a brief schema description or using a system message that lists table structures (as shown in Section 1). This approach is essentially providing a retrieval-augmented prompt – all the info needed is in the model’s input, so it doesn’t have to guess column names or relationships ([Natural Language Queries to SQL with Code-Llama | E2E Cloud](https://docs.e2enetworks.com/docs/tir/Inference/Tutorials/Natural_Language_Queries_to_SQL_with_Code_Llama/#:~:text=It%20should%20be%20ensured%20that,it%E2%80%99s%20output%20is%20given%20below)).
* **Use Few-Shot Examples:** One powerful technique is to include one or two example NL-to-SQL pairs in the prompt (few-shot learning) to demonstrate the correct format and use of joins. Research has found that a few in-domain examples can significantly improve accuracy – sometimes even outperforming models fine-tuned on small datasets ([[2204.00498] Evaluating the Text-to-SQL Capabilities of Large Language Models](https://ar5iv.org/abs/2204.00498#:~:text=on%20the%20Spider%20benchmark%3B%20we,shot%20examples)). For instance, you might add to the prompt: *“Example: Question: 'How many transactions did branch X have?' → SQL: SELECT b.branch\_name, COUNT(\*) FROM branches AS b JOIN transactions AS t ON b.branch\_id=t.branch\_id WHERE b.branch\_name='X';”*. A carefully chosen example that involves a join and uses the correct syntax can prime the model to do the same for the user's query. Ensure your examples reflect common pitfalls (like needing a join or a GROUP BY) so the model sees how to handle them. Keep the examples within the same schema so the model doesn’t confuse table names. Few-shot prompting is especially useful if you notice the model making a recurring mistake – you can create a demonstration that corrects it.
* **Explicit Column Disambiguation:** In multi-table queries, instruct (or demonstrate) the use of table aliases and explicit column references. This prevents any confusion if different tables have columns with the same name and generally leads to clearer SQL. For example, always prefix columns with the table alias (e.g., b.branch\_name instead of just branch\_name). This habit can be encouraged by the prompt or examples. It not only helps the model avoid ambiguity but also often catches join issues (because if you write transactions.transaction\_amount and branches.branch\_name in the SELECT, the model will realize a JOIN is needed). Many SQL generation errors in LLMs come from using a column name without specifying which table it’s from, leading to either syntax errors or unintended Cartesian products. By consistently using aliases in the prompt and example, the model will mimic that style.
* **Iterative Refinement (Feedback Loop):** In a production scenario, you can implement a feedback loop where the model’s output is validated (by the database or a parser) and any errors are fed back for correction. For example, if the model produces a SQL that fails to execute, capture the error message (e.g., “column X does not exist”) and supply it to the model in a follow-up prompt like: *"The previous query resulted in an error: column X does not exist. Please fix the SQL query."* Code Llama models are capable of understanding such feedback and adjusting the query accordingly. While this is outside the single-turn prompt, it’s a strategy to significantly improve reliability. Essentially, the model can serve as its own debugger when given the chance to correct mistakes. This aligns with the idea of *including user feedback or session history* to handle complex queries ([Prompt Engineering for a Better SQL Code Generation With LLMs | by Raghid Bsat | Dataminded | Medium](https://medium.com/datamindedbe/prompt-engineering-for-a-better-sql-code-generation-with-llms-263562c0c35d#:~:text=2,history)) – you can treat the database’s error as feedback. In practice, you’d automate this: run the query, and if an error occurs, prompt the model again with the error context. This can resolve issues like missing JOINs or misnamed columns in the second attempt.
* **Clarify Ambiguities in Natural Language:** Sometimes users ask questions that are inherently ambiguous (e.g., “Show me the transactions for last year” – does that mean sum of amounts, count of transactions, or list of all transactions?). The model might guess and be wrong. One way to mitigate this is to have the prompt encourage the model to assume a reasonable interpretation *or* ask for clarification. However, since we want the model to output SQL directly, it's better to handle ambiguity upstream. If possible, preprocess the NL query to a more precise form or add notes in the system prompt like: *“If a request is ambiguous, choose a reasonable interpretation and proceed.”* In our scenario, for instance, “transactions totaling over 10000 USD this month” could mean sum of amounts per branch over 10000. The prompt could explicitly frame it: *“User asks for branches with total transaction\_amount > 10000 in the current month – so you should sum transactions by branch and filter.”* This kind of hint ensures the model doesn’t interpret it as looking for individual transactions over 10000. Essentially, use the prompt to **inject subtle clarifications** for anything that might be interpreted in multiple ways. This reduces the chance of logical errors even if the question was a bit vague.
* **Leverage Model Documentation and Patterns:** The LLaMA 2 and CodeLlama training documentation notes that the instruct models are tuned to follow natural language instructions carefully and produce helpful responses ([Exploring Different Capabilities of CodeLlama](https://www.analyticsvidhya.com/blog/2024/07/different-capabilities-of-codellama/#:~:text=,CodeLlama%20can%20be%20used%20to)) ([Exploring Different Capabilities of CodeLlama](https://www.analyticsvidhya.com/blog/2024/07/different-capabilities-of-codellama/#:~:text=Here%20are%20examples%20that%20illustrate,providing%20relevant%20responses%20to%20prompts)). Take advantage of this by phrasing the prompt in a polite but direct instructional manner (as shown in Section 1). Also, the model has likely seen examples of SQL and code in its training. It might have learned common patterns (e.g., SELECT ... FROM ... JOIN ... ON ... WHERE ...). You can nudge it by including partial skeletons if needed. One trick community users have tried is providing a “query plan” or comment in the prompt, like: *"-- Plan: need to join branches and transactions, then aggregate"* followed by the actual SQL generation. CodeLlama might pick up on the commented plan and follow it. However, be cautious – the model might include the comment in the SQL if not instructed properly. A safer approach is simply to enumerate steps in plain language in the system prompt (not as part of the SQL code). For example: *"Steps to solve: (1) join branches and transactions, (2) group by branch, (3) filter sum > 10000. Now write the SQL."* This engages the model’s chain-of-thought without asking it to output the chain. It leverages the model’s ability to internally reason ([Prompt Engineering for a Better SQL Code Generation With LLMs | by Raghid Bsat | Dataminded | Medium](https://medium.com/datamindedbe/prompt-engineering-for-a-better-sql-code-generation-with-llms-263562c0c35d#:~:text=requires%20contextual%20understanding%20and%20inference,produced%20an%20invalid%20SQL%20statement)) but keeps the final output clean.
* **Testing and Refinement:** Finally, treat this as an iterative process. Monitor the model’s outputs on a variety of user questions and note where it fails. You might discover it consistently mishandles a certain phrasing or always forgets a GROUP BY clause for aggregation questions. Use this insight to refine your prompt. For instance, if it often omits GROUP BY, add a line in the instruction: *"Remember to use GROUP BY when aggregating."* If it confuses transaction\_type and transaction\_amount, maybe the schema description can be expanded (e.g., *"transactions.transaction\_type is a category (like 'deposit' or 'withdrawal'), transaction\_amount is numeric."*). The prompt can be optimized over time to include these clarifications. This is an application of prompt engineering: analyzing model mistakes and adjusting the prompt or examples to preempt them ([Prompt Engineering for a Better SQL Code Generation With LLMs | by Raghid Bsat | Dataminded | Medium](https://medium.com/datamindedbe/prompt-engineering-for-a-better-sql-code-generation-with-llms-263562c0c35d#:~:text=tool%20sounds%20amazing%20and%20training,information%20directly%20in%20the%20prompt)).
* **Consider Fine-Tuning for Further Improvement:** If prompt tuning and decoding settings are still not giving the desired accuracy, note that specialized fine-tuned models for text-to-SQL exist and can be used as a next step. For example, the **NSQL-Llama-2-7B** model was fine-tuned on text-to-SQL data and showed a 15-point accuracy improvement over base Llama 2 on complex benchmarks ([NSQL-Llama-2-7b: Advancing SQL Foundation Models for Personalized Analytics - Numbers Station](https://www.numbersstation.ai/nsql-llama-2-7b/#:~:text=%23%23%20This%20blog%20introduces%20NSQL,4%20on%20standard%20SQL%20benchmarks)) ([NSQL-Llama-2-7b: Advancing SQL Foundation Models for Personalized Analytics - Numbers Station](https://www.numbersstation.ai/nsql-llama-2-7b/#:~:text=Technical%20Analysis)). Similarly, an open project by MotherDuck and NumbersStation created a 7B model fine-tuned for SQL (using CodeLlama as base) that achieved very strong performance on benchmarks ([Prompt Engineering for a Better SQL Code Generation With LLMs | by Raghid Bsat | Dataminded | Medium](https://medium.com/datamindedbe/prompt-engineering-for-a-better-sql-code-generation-with-llms-263562c0c35d#:~:text=SQL%20query%20generator,additional%20context%20to%20the%20model)). Fine-tuning your model on a labeled NL→SQL dataset that reflects your schema and queries can greatly enhance join handling and reduce errors. This goes beyond prompt tweaks – it actually teaches the model the correct patterns. In a production setting where errors must be minimal, investing in fine-tuning (or using a community fine-tuned model) might be worthwhile if the base model plus prompting isn't sufficient. That said, for many use cases, a well-crafted prompt on CodeLlama 7B Instruct can already produce excellent results, as CodeLlama has demonstrated strong zero-shot text-to-SQL abilities ([Exploring Different Capabilities of CodeLlama](https://www.analyticsvidhya.com/blog/2024/07/different-capabilities-of-codellama/#:~:text=Code,executed%20on%20an%20existing%20database)) ([Exploring Different Capabilities of CodeLlama](https://www.analyticsvidhya.com/blog/2024/07/different-capabilities-of-codellama/#:~:text=This%20query%20will%20result%20in,)) (often outperforming even larger models on code tasks).

By applying these techniques – robust prompt design, careful parameter tuning, and iterative refinement – you can harness LLaMA Code 7B Instruct to reliably convert natural language questions into correct PostgreSQL SQL queries. The key is to reduce the model’s uncertainty: give it all the info it needs (schema, clarifications) and constrain its generation just enough that it sticks to correct syntax. With practice, you'll find the model can handle a wide range of queries, from simple selections to complex joins, making it a valuable component in a natural language to SQL pipeline for production use.

**Sources:**

* Meta AI, *Code Llama 7B Instruct* – Model card and usage guidelines ([codellama/CodeLlama-34b-hf · Hugging Face](https://huggingface.co/codellama/CodeLlama-34b-hf" \l ":~:text=sequences%20%3D%20pipeline%28%20%27import%20socket,eos_token_id%2C%20max_length%3D200)) ([Exploring Different Capabilities of CodeLlama](https://www.analyticsvidhya.com/blog/2024/07/different-capabilities-of-codellama/#:~:text=,CodeLlama%20can%20be%20used%20to))
* Analytics Vidhya, *Using Code Llama for Text-to-SQL* – Prompt format with schema and example SQL generation ([Exploring Different Capabilities of CodeLlama](https://www.analyticsvidhya.com/blog/2024/07/different-capabilities-of-codellama/#:~:text=prompt%20%3D%20f,question%7D%3A%20%60%60%60)) ([Exploring Different Capabilities of CodeLlama](https://www.analyticsvidhya.com/blog/2024/07/different-capabilities-of-codellama/#:~:text=This%20query%20will%20result%20in,))
* E2E Networks, *Natural Language Queries to SQL* – Challenges in NL→SQL and importance of dialect specificity ([Natural Language Queries to SQL with Code-Llama | E2E Cloud](https://docs.e2enetworks.com/docs/tir/Inference/Tutorials/Natural_Language_Queries_to_SQL_with_Code_Llama/#:~:text=1)) ([Natural Language Queries to SQL with Code-Llama | E2E Cloud](https://docs.e2enetworks.com/docs/tir/Inference/Tutorials/Natural_Language_Queries_to_SQL_with_Code_Llama/#:~:text=It%20should%20be%20ensured%20that,it%E2%80%99s%20output%20is%20given%20below))
* Medium – *Prompt Engineering for SQL with LLMs* by R. Bsat – Emphasizing schema inclusion and prompt clarity ([Prompt Engineering for a Better SQL Code Generation With LLMs | by Raghid Bsat | Dataminded | Medium](https://medium.com/datamindedbe/prompt-engineering-for-a-better-sql-code-generation-with-llms-263562c0c35d#:~:text=1)) ([Prompt Engineering for a Better SQL Code Generation With LLMs | by Raghid Bsat | Dataminded | Medium](https://medium.com/datamindedbe/prompt-engineering-for-a-better-sql-code-generation-with-llms-263562c0c35d#:~:text=tool%20sounds%20amazing%20and%20training,information%20directly%20in%20the%20prompt))
* Numbers Station – *NSQL-Llama2-7B Announcement* – Benefits of training and examples of improved join handling ([NSQL-Llama-2-7b: Advancing SQL Foundation Models for Personalized Analytics - Numbers Station](https://www.numbersstation.ai/nsql-llama-2-7b/#:~:text=we%20show%20that%20NSQL,%E2%80%8D)) ([Prompt Engineering for a Better SQL Code Generation With LLMs | by Raghid Bsat | Dataminded | Medium](https://medium.com/datamindedbe/prompt-engineering-for-a-better-sql-code-generation-with-llms-263562c0c35d#:~:text=SQL%20query%20generator,additional%20context%20to%20the%20model))
* Rajkumar et al., *Evaluating Text-to-SQL via Codex* – Few-shot prompting effectiveness for SQL generation ([[2204.00498] Evaluating the Text-to-SQL Capabilities of Large Language Models](https://ar5iv.org/abs/2204.00498#:~:text=on%20the%20Spider%20benchmark%3B%20we,shot%20examples))
* Implementation example: *CodeLlama-7B text-to-SQL demo* (HuggingFace) – showing system prompt with schema and using greedy decoding ([finetune-LLMs/app.py at main · Pyligent/finetune-LLMs · GitHub](https://github.com/Pyligent/finetune-LLM/blob/main/app.py#:~:text=schema%20%3D%20%27You%20are%20an,school%20VARCHAR%2C%20last_occ_championship%20VARCHAR))