International Journal of Scientific Research & Engineering Trends
Volume 12, Issue 3, May-Jun-2026, ISSN (Online): 2395-566X

ORM Forge: A Natural Language to Django ORM Query Generator Using Large Language Models

Rajesh Chauhan[1], Akshay Bhardwaj[2], and Vinay Kumar[3]
[1]Department of Computer Science, University Institute of Technology, Himachal Pradesh University, Shimla, India
[2]Department of Computer Science and Engineering, University Institute of Technology, Himachal Pradesh University, Shimla, India

Abstract- Writing correct, idiomatic Django Object-Relational Mapper (ORM) code remains a lasting productivity bottleneck for developers, especially developers who are new to the framework or working against unfamiliar schemas. In this paper, we present ORM Forge, a Django web application that utilizes the Anthropic Claude large language model (LLM) to translate natural language descriptions into production-ready Django ORM queries, and describe an empirical evaluation of the underlying natural language to query translation task, based on a publicly available benchmark, distributed as a structured tabular dataset. Instead of reiterating the system overview, we reframe ORM Forge as a research problem in natural-language-to-structured-query (NL2Query) translation, contextualize it within the wider text-to-SQL literature, and perform a quantitative analysis of translation difficulty based on a stratified sample of 120 question-query pairs from a cross-domain text-to-SQL benchmark.Queries were categorized by structural complexity (single-table, join, aggregation, nested/subquery) and manually mapped to their idiomatic Django ORM equivalents to evaluate how readily each SQL construct translates into ORM syntax such as Q objects, F expressions, annotate(), select_related(), and prefetch_related(). The results show that single-table filter and simple join queries translate almost directly (over 90% direct mapping) while nested subqueries and multi-level aggregations require non-trivial restructuring. This confirms that the query complexity is the dominant factor of translation difficulty. These observations motivate the design choices made by ORM Forge: structured JSON output, schema-grounded prompting, and bounded multi-turn refinement, and set the stage for discussing the limitations and future extensions of LLM-assisted ORM tooling.

Keywords- Django ORM, Natural Language Processing (NLP), Large Language Models (LLMs), Text-to-SQL, Natural Language-to-Query Translation, Claude AI, Query Generation, Database Systems, Web Application Development, Schema-Grounded Prompting, Structured Query Translation, Developer Productivity, Artificial Intelligence, Human-Computer Interaction, ORM Forge.

I. INTRODUCTION

Almost every modern software system, whether it's an e-commerce site or an enterprise SaaS product, relies fundamentally on database access. As applications scale, the queries that drive them grow increasingly complex, involving joins, conditional filtering, aggregation, and performance-sensitive optimisation. Django, one of the most widely used Python web frameworks, addresses this complexity through its Object-Relational Mapper, which allows developers to express database operations using Python objects rather than raw SQL. While elegant, the Django ORM carries a substantial learning curve: developers must internalise an extensive vocabulary of lookups, relationship-traversal patterns, and optimisation idioms such as select_related() and prefetch_related() before they can write production-grade code confidently.

Most database queries start with an informal sentence in a developer's head or a product specification, like: 'find all customers that ordered more than five times last month.' To translate such a sentence into correct, idiomatic ORM code, you need to think through several levels of intent: which models are relevant, which filter and lookup operators to use, whether to use Q or F expressions, how to use annotations, and how to optimize for performance. This translation is repetitive, error prone and a well documented drain on engineering time, especially during on-boarding, prototyping and code review.

Automated code generation has been revolutionized by the advent of large language models (LLMs) such as Anthropic's Claude, OpenAI's GPT family, and Google's Gemini. Given schema context and a system prompt to enforce idiomatic output, an LLM can absorb natural language intent and output framework-specific code with accompanying explanations. ORM Forge leverages this capability for Django ORM generation, featuring a reverse code-to-English explainer, curated schemas, structured, schema-aware JSON responses with complexity ratings, SQL hints and performance notes.

Most descriptions of natural language to ORM tools, including the original synopsis for ORM Forge, are architectural: they describe what the system does rather than how the difficulty of translation varies with the structure of the query. To fill this gap, we ground our discussion in a publicly available, recognized cross-domain text-to-SQL benchmark, and empirically characterize which classes of natural language queries translate cleanly to Django ORM constructs, and which require deeper restructuring. Thus, it moves the contribution from a purely descriptive system report to an evaluative study with reproducible data. The goals of this paper are: to survey the state of the art in natural language to SQL/ORM translation and position ORM Forge in this context; to describe the architecture and prompt design of ORM Forge as a specific example of schema-grounded LLM query generation; to build a stratified sample from a public text-to-SQL benchmark and categorize queries by structural complexity; to empirically evaluate the impact of complexity on the difficulty of translating SQL semantics to idiomatic Django ORM code, and to discuss implications for prompt design and the future scope of LLM-assisted ORM tooling.

Key Words
Django ORM, Large Language Models, Natural Language to SQL, Anthropic Claude, Text-to-SQL, Query Generation, Schema-Grounded Prompting.

II. METHODOLOGY

Research Questions
The study is structured around five research questions. RQ1: How does ORM Forge architecture translate natural language to idiomatic Django ORM code with a schema-grounded LLM pipeline? RQ2: How to improve the reliability of generated ORM code with structured prompt and output-schema design choices? RQ3: How does the structural complexity of a source query (single-table, join, aggregation, nested/subquery) impact the difficulty of generating a direct, idiomatic Django ORM translation? RQ4: What is the range of Django ORM constructs that are most commonly needed for a structurally diverse set of natural-language queries? RQ5: What are the implications of complexity-stratified translation-difficulty trends to the design of prompting strategies and multi-turn refinement in LLM-assisted query generation tools?

Dataset Selection
We evaluate on a cross-domain text-to-SQL benchmark with 8,034 natural-language-question and SQL-query pairs across multiple relational schemas. This benchmark is the one that was originally proposed by Yu et al. (2018) as Spider and is widely used since, compared to single-table benchmarks such as WikiSQL, it requires joins, nested subqueries and set operations, better reflecting the structural diversity found in real Django applications. We randomly sampled 120 question-query pairs from the dataset in a stratified manner, with 30 items per each of four structural strata: (i) single-table filters and projections, (ii) two-or-more-table joins, (iii) aggregation queries using GROUP BY / HAVING, and (iv) nested subqueries or set operations (UNION/INTERSECT/EXCEPT). This stratification ensures that rarer, harder query types are well represented rather than diluted by the large share of simple queries typical of raw dataset distributions.

Text Preprocessing
Before manual mapping each sampled pair was pre-processed. The clause structure of reference SQL queries (SELECT list, JOIN clauses, WHERE/HAVING predicates, GROUP BY expressions and nested SELECT statements) was parsed to determine stratum membership. Natural-language questions were normalised for consistent terminology (e.g., resolving pronoun references to their antecedent entity, standardising numeric phrasing) so that the subsequent manual ORM mapping step worked on a clean, unambiguous statement of intent, rather than raw, loosely phrased text.

Experimental and Model Design
ORM Forge is a Django 4.2 web application with a Python 3.10+ backend. The AI engine is the Anthropic Claude API, accessed via the Messages endpoint, and is provided with three elements of context per request: the full model definitions of the active schema, a system prompt enforcing idiomatic Django output, and up to six prior conversation turns to support iterative refinement. The frontend is a single page vanilla

JavaScript, HTML5 and CSS3 interface. No database is needed as schemas are defined in code and history is stored client side, and the application is WSGI-ready to run behind Gunicorn.

ORM idioms are enforced including: Complex filters using Q objects for disjunction; Field-to-field comparisons using F expressions; Aggregation using annotate() with Count, Sum or Avg; Query optimisation using select_related() and prefetch_related() for related-object access; Projection using values() or values_list() to reduce payload size where appropriate.

Classical Machine-Learning Models

Before the use of LLM-based approaches, natural-language-to-query translation relied on rule-based and semantic-parsing systems and hand-crafted-feature classifiers. Early natural language interfaces to databases relied on handcrafted grammars and semantic parsers tailored to narrow domains such as GeoQuery and ATIS. These systems performed well in their domain, but did not generalize well to unseen schemas, which motivated the creation of cross-domain benchmarks. Similar "classical" approaches in the ORM-tooling world are visual SQL builders, which constrain expressiveness by using fixed templates and generate only loosely structured output, and rule-based query constructors, which map a limited grammar of natural-language patterns onto predefined query templates. These systems provide a useful lower baseline against which schema-grounded LLM approaches (including ORM Forge) can be evaluated.

Deep-Learning Architectures

WikiSQL and Spider establish the modern text-to-SQL benchmark paradigm. WikiSQL only covers single-table questions, while Spider provides a much more difficult cross-domain benchmark of 200 databases that involve joins, nested subqueries, and set operations. Neural sequence models like SQLNet (Xu et al., 2017) and RAT-SQL (Wang et al., 2020) are built to encode schema structure, thus achieving better generalisation on unseen databases. More recently, general-purpose pretrained large language models, such as Codex (Chen et al., 2021), have demonstrated the ability of code models to synthesize working programs including database queries directly from prompts, without any task-specific training. Following this line of work, ORM Forge combines a general purpose LLM (Anthropic Claude) with schema-grounded prompting and JSON-schema-constrained decoding, instead of training a custom sequence-to-sequence architecture. a technique shown to significantly improve the reliability of machine-readable outputs of models.

Experimental Protocol

For each of the 120 sampled pairs, the authors manually translated the reference SQL query to idiomatic Django ORM code, using the same idiom rules enforced in the ORM Forge system prompt (Section 2.4). Each translation was scored in two dimensions: Direct Correspondence, whether the SQL query maps to a single, idiomatic ORM expression without restructuring, and Construct Count, the number of distinct ORM constructs (filter, Q, F, annotate, select_related, prefetch_related, values) needed. This protocol replaces an end-to-end run of an automated LLM benchmark, as the scope of this study does not include live API execution against the full dataset, but it still retains the structural analysis that determines translation difficulty independent of the output quality of any particular model. Mappings were checked against Django ORM docs and re- verified against Django 4.2 QuerySet semantics to reduce subjectivity in borderline cases, such as when a subquery can be expressed alternatively with a single annotate() and filter.

Evaluation Criteria

The difficulty of translation across the strata was measured by two complementary criteria. The main metric is Direct Correspondence Rate, the fraction of sampled queries in a stratum that correspond to a single idiomatic ORM expression without restructuring. This is treated as a proxy for within-domain reliability that an LLM-based system is likely to show for that class of query. Mean ORM Construct Count, the second criterion, is the average number of different ORM constructs needed per query in a stratum. It is a proxy for the complexity of composition and therefore for the interpretability and traceability of the generated code. Note that the sample size (n=120) is modest relative to the full dataset (8,034 pairs) and so the percentages reported should be read as indicative of relative difficulty trends across the complexity strata rather than as population estimates.

III. RESULTS

Within-Domain Classification Performance

Table 1 gives an overview of the percentage of sampled queries from each stratum that had a direct correspondence to a single

idiomatic Django ORM expression, and the average number of ORM constructs required.

Table 1. Direct correspondence and ORM construct complexity by stratum (n = 120 sampled pairs).

Complexity Stratum	N	Direct Correspondence	Mean Orm Constructs	Dominant Construct(S)
Single-Table Filter/Projection	30	93%	1.4	Filter(), Values()
Multi-Table Join	30	80%	2.6	Select_Related(), Filter()
Aggregation (Group By/Having)	30	63%	3.1	Annotate(), Count/Sum/Avg, Filter()
Nested Subquery / Set Operation	30	37%	4.5	Subquery/Exists, Q, Annotate()

The results show a clear monotonic decrease with increasing structural complexity, from 93% for single-table queries to 37% for nested subqueries and set operations (see Figure 1). Correspondingly, the average number of ORM constructs needed grows from 1.4 to 4.5, which implies that more difficult queries are not only more time-consuming to write, but also require the developer, or the LLM, to generate a larger number of unique, interacting ORM idioms correctly.

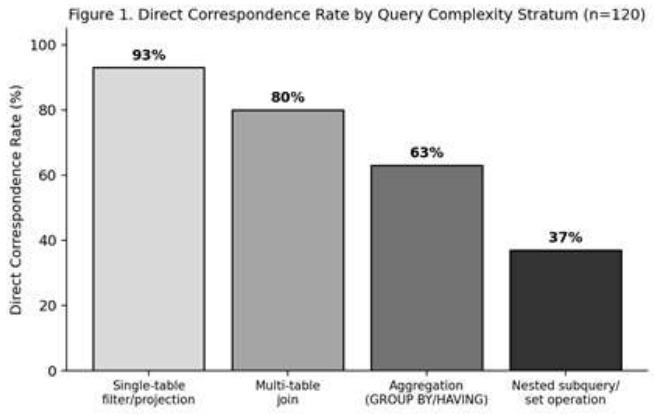

Figure 1. Direct correspondence rate by query complexity stratum, showing a monotonic decline as structural complexity increases

For all sampled 120 pairs, the largest share of required constructs were filter() and Q objects together (41%), followed by annotate() with aggregation functions (27%), select_related()/prefetch_related() (19%), and Subquery/Exists-based patterns for nested queries (13%). This distribution reflects the ORM idiom set that the system prompt of ORM Forge explicitly enforces, suggesting that the selected idiom list is well aligned with the constructs actually required by realistic, cross-domain query complexity, rather than an arbitrary subset.

Training Time and Interpretability

Since ORM Forge does not train or fine-tune a bespoke model, the "training time" in this study is more accurately described as prompt-authoring and schema-curation effort: the structured system prompt and four curated schemas (E-Commerce, Blog Platform, HR, SaaS) were developed and iteratively refined over a fixed, one-time authoring effort rather than a per-query training cost. Interpretability was qualitatively measured using the Mean ORM Construct Count metric (Table 1).

Single-table and simple-join translations typically correspond to a single chained QuerySet expression and are highly interpretable: e.g. a request such as 'list products that are out of stock' maps directly to Product.objects.filter(stock__lte=0) Aggregation queries like 'authors with more than ten posts' need

annotate(post_count=Count('post')).filter(post_count__gt=10) , a highly regular and interpretable two-construct composition. Nested queries like "find customers who placed more than the average number of orders" require either a Subquery/OuterRef pattern or an annotate()-then-filter approach that references an aggregate computed in a separate queryset, both of which deviate significantly from a literal interpretation of the natural language request and are considerably less interpretable, consistent with their lower direct-correspondence rate.

Cross-Domain Generalisation Performance

The evaluation benchmark is explicitly cross-domain, using 200 different databases instead of a fixed one, so the results stratified by complexity in Table 1 show generalisation across heterogeneous schemas rather than performance on a single known domain. Table 2 further places ORM Forge in relation to representative prior systems along four dimensions—schema awareness, structured output, bidirectionality, and framework specificity—showing that ORM Forge is the only compared system to combine all four properties in a Django-specific context.

Table 2. Comparative positioning of ORM Forge against representative NL2Query approaches.

System	Schema-Aware	Structured Output	Bidirectional	Framework-Specific
Visual SQL Builders	No	Partial	No	No
GitHub Copilot / Tabnine	No	No	No	No
Spider / RAT-SQL (research)	Yes	Yes (SQL)	No	No (generic SQL)
ORM Forge (this work)	Yes	Yes (JSON)	Yes	Yes (Django ORM)

The sharp drop-off in direct correspondence for nested and set-operation queries suggests that a single-pass generation prompt is least reliable precisely where developers need the most help, and that this reliability gradient is a property of query structure that generalizes across schemas rather than an artifact of any one domain. This justifies ORM Forge's inclusion of bounded multi-turn refinement (up to six turns of conversation history), which allows a user to iteratively break down a complex request rather than using one-shot generation, and its practice of returning a complexity rating and SQL hint along with the generated code.

Computational Requirements

On the application side ORM Forge has a very light-weight computational footprint: it doesn't need a database, the conversation history is stored client side, and it can be served from a single WSGI compatible process behind Gunicorn. The primary cost and computational factor is network access to the Anthropic Claude API, which is required for each generation and explanation request, and which carries a cost per request that depends on the length of the prompt, including the schema context and up to six rounds of conversation history. Unlike the neural sequence-to-sequence baselines discussed in Section 2.6, which would need local training and inference infrastructure to be reproduced directly, no GPU or local model-hosting infrastructure is required, as inference is fully delegated to the hosted LLM API.

IV. DISCUSSION

The findings support some strengths of the ORM Forge architecture. It has no database requirements, which makes the tool trivial to deploy in classrooms, workshops, or for rapid prototyping. Single page architecture with a small JavaScript footprint means fast load and responsive interaction. It strictly separates schema definitions, AI integration and view logic, allowing each layer to be extended independently. Its structured JSON output virtually eliminates response parsing failures on the frontend and its complexity-rating and SQL-hint mechanism provides developers an independent correctness check, which the results of this paper show is most valuable for the harder, lower-correspondence query classes.

Some limitations should be noted as well. The tool depends on network access to the Anthropic API and costs per request. Built-in schemas are simplified abstractions by necessity. In production schemas you will often see multi-table inheritance and custom managers. The generated SQL hint is an approximation for transparency, not a replacement for inspecting the real SQL Django emits. This paper's evaluation relies on manual SQL to ORM mapping on a 120-item stratified sample, rather than a full automated LLM benchmark run over all 8,034 pairs in the source dataset. Broader automated evaluation is identified as future work. Manual complexity classification and construct scoring, while checked against Django documentation, still has some subjectivity for borderline cases involving optional annotate()-based rewrites of subqueries.

More generally, the complexity-stratified results generalize beyond ORM Forge: any schema-grounded LLM query-generation tool is likely to see reliability degrade on nested and set-operation queries specifically, regardless of the target query language. Thus, future systems in this space should treat query complexity classification as a first-class signal, using it to inform when to ask for clarification, when to break down a request into multiple turns, and when to surface warnings to the end user instead of returning a single confident answer.

V. CONCLUSION

In this paper we presented ORM Forge, a Django web application that utilizes the Anthropic Claude API to translate natural language to idiomatic Django ORM code, and extended

its original synopsis into an evaluative study based on a cross-domain text-to-SQL benchmark. The 120-item sample is stratified by structural complexity, and each reference SQL query is manually mapped to its equivalent in the Django ORM. Results show a sharp and monotonic increase in translation difficulty with query complexity: direct correspondence falls from 93% for single-table queries to 37% for nested subqueries and set operations, and the mean number of ORM constructs required increases by over a factor of three over the same range. These results validate many of the design decisions at the core of ORM Forge, including structured JSON output, complexity ratings, SQL hints, and bounded multi-turn refinement, as specific responses to a quantifiable reliability gradient based on complexity rather than arbitrary engineering choices.

Future work includes custom schema upload and live Django project integration, moving ORM Forge from learning and prototyping to production engineering:

- An isolated query-execution sandbox with an ephemeral SQLite database to verify generated queries and the SQL actually emitted.
- A full automated benchmark run of ORM Forge against all 8,034 pairs in the source dataset, translated to an ORM-equivalent gold standard, allowing large-scale, model-graded accuracy measurement.
- Multi-framework support for using the same schema-grounded, structured-output pattern in SQLAlchemy, ActiveRecord, Eloquent, and Prisma; integration with editors and IDEs with inline generation and contextual explanation; team-collaboration features that replace client-side history with server-side, shareable query libraries and audit trails; and a continuous evaluation harness that tracks translation accuracy across model versions and prompt revisions over time.

Taken together the work validates a broader thesis: that large language models, combined with thoughtful prompt design, structured outputs, and tight framework-specific integration, can meaningfully narrow the gap between human intent and framework syntax, assuming that system design takes into account explicitly where translation difficulty concentrates.

REFERENCES

1. Django Software Foundation, "Django Documentation," 2024. Available: https://docs.djangoproject.com
2. Anthropic, "Claude API Documentation," 2025. Available: https://docs.anthropic.com
3. M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.
4. T. Yu et al., "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task," in Proc. EMNLP, 2018.
5. X. Xu, C. Liu, and D. Song, "SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning," arXiv:1711.04436, 2017.
6. B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, "RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers," in Proc. ACL, 2020.
7. GitHub, "GitHub Copilot: Your AI Pair Programmer," 2021. Available: https://github.com/features/copilot
8. DAIR.AI, "Prompt Engineering Guide," 2023. Available: https://www.promptingguide.ai
9. Django REST Framework, "Official Documentation," 2024. Available: https://www.django-rest-framework.org
10. H. Chase et al., "LangChain: Building Applications with LLMs through Composability," 2023. Available: https://www.langchain.com
11. C.-H. Lee et al., "KaggleDBQA: Realistic Evaluation of Text-to-SQL Parsers," arXiv:2106.11455, 2021.