Towards Small Language Models for Security Query Generation in SOC Workflows

Saleha Muzammil*
evz4sc@virginia.edu
University of Virginia
United States

Rahul Reddy*†
dqb5ty@virginia.edu
University of Virginia
United States

Vishal Kamalakrishnan
vishkamalk@gmail.com
University of Virginia
United States

Hadi Ahmadi
hadi@corvic.ai
Corvic
United States

Wajih Ul Hassan
hassan@virginia.edu
University of Virginia
United States

Abstract

Analysts in Security Operations Centers routinely query massive telemetry streams using Kusto Query Language (KQL). Writing correct KQL requires specialized expertise, and this dependency creates a bottleneck as security teams scale. This paper investigates whether Small Language Models (SLMs) can enable accurate, cost-effective natural-language-to-KQL translation for enterprise security. We propose a three-knob framework targeting prompting, fine-tuning, and architecture design. First, we adapt existing NL2KQL framework for SLMs with lightweight retrieval and introduce error-aware prompting that addresses common parser failures without increasing token count. Second, we apply LoRA fine-tuning with rationale distillation, augmenting each NLQ-KQL pair with a brief chain-of-thought explanation to transfer reasoning from a teacher model while keeping the SLM compact. Third, we propose a two-stage architecture that uses an SLM for candidate generation and a low-cost LLM judge for schema-aware refinement and selection. We evaluate nine models (five SLMs and four LLMs) across syntax correctness, semantic accuracy, table selection, and filter precision, alongside latency and token cost. On Microsoft's NL2KQL Defender Evaluation dataset, our two-stage approach achieves 0.987 syntax and 0.906 semantic accuracy. We further demonstrate generalizability on Microsoft Sentinel data, reaching 0.964 syntax and 0.831 semantic accuracy. These results come at up to 10x lower token cost than GPT-5, establishing SLMs as a practical, scalable foundation for natural-language querying in security operations.

1 Introduction

The evolving cybersecurity landscape has increased attack complexity and reshaped how analysts mitigate threats. In Security Operations Centers (SOCs), analysts face overwhelming volumes of event logs, network traffic, and threat intelligence feeds [6], receiving on average 5,000 alerts per day via SIEM systems and up to 100,000 in extreme cases [4]. To investigate these massive logs, they rely on query languages such as Sigma, Elastic EQL, and KQL [2, 10, 28], but translating natural-language intent into correct queries over large, evolving schemas remains a major bottleneck.

KQL, introduced by Microsoft in 2017 [19], is the most widely adopted. Integrated into Microsoft Sentinel and Defender, it is a domain-specific, read-only language that lets analysts parse millions of log rows efficiently and extract relevant events. However, its expressiveness and non-trivial semantics make authoring accurate queries challenging for non-experts, especially during time-sensitive investigations.

```
DeviceNetworkEvents
| where Timestamp >= ago(7d)
| where ActionType == 'ConnectionSuccess'
| summarize arg_max(Timestamp, LocalIP) by DeviceId
| where LocalIP == "89.12.55.1"
| project DeviceId
```

Listing 1: Example KQL query that retrieves device IDs which last connected from IP address 89.12.55.1 within the last 7 days. The query references multiple columns in the `DeviceNetworkEvents` table schema, including `Timestamp`.

In the era of LLMs, automated KQL generation can further accelerate investigations by producing precise queries aligned with an analyst's intent. Beyond code generation, LLMs have been applied across cybersecurity tasks, such as log analysis and penetration testing [8, 23]. They have also been exploited for offensive purposes, including phishing and ransomware planning [13], while models like GPT-3.5 and GPT-4 show strong defensive potential in areas such as scam detection [17]. However, LLM-centric pipelines for NLQ-to-KQL translation can be costly, latency-sensitive, and difficult to deploy under enterprise data-governance constraints.

Unfortunately, LLMs also incur drawbacks, including higher latency, memory demands, and operational costs. Moreover, depending on the use case, they may be unsuitable for certain tasks [7]. As an alternative, we introduce SLMs, defined following [7] as language models that (i) can run on common consumer devices with practical inference latency, (ii) are not LLMs, and (iii) have at most 10 billion parameters. SLMs offer lower latency, memory, computational, and operational costs while still achieving accuracy comparable to LLMs in some domain-specific tasks [7]. Despite this promise, there has been no systematic study of SLMs for NLQ-to-KQL translation in realistic, schema-rich settings.

This work fills that gap through three orthogonal enhancement knobs (prompting, fine-tuning, and two-staged architecture design) and a comprehensive empirical study. We provide the first systematic evaluation of SLMs for NLQ-to-KQL translation across multiple accuracy metrics, latency, and cost, establishing clear baselines against representative LLMs. We adapt the NL2KQL architecture

*Equal contribution; authors sorted alphabetically by last name.
†Not responsible for any DeepSeek-related experiments or analysis in this paper.

for SLMs by replacing heavy components with lightweight alternatives while preserving the prompting, retrieval, and refinement structure so that results are both comparable and cost efficient. We also introduce error-aware prompting based on common KQL parser failures, which improves syntactic and semantic correctness without increasing token count.

We further explore LoRA fine-tuning [15] with rationale distillation to transfer reasoning from a teacher LLM into an SLM. LoRA trains only low-rank adapter parameters while freezing the original model weights, enabling efficient fine-tuning on NLQ-KQL pairs. To embed reasoning capabilities, each training example is augmented with a short chain-of-thought explanation produced by the teacher model, followed by the target KQL. This rationale-augmented setup allows the SLM to learn intermediate reasoning steps in addition to final outputs, strengthening its ability to handle structured code generation tasks without increasing model size.

Beyond prompting and fine-tuning, our core contribution is a two-stage SLM-Oracle architecture for KQL generation. The first stage uses an SLM to efficiently generate candidate queries, while the second employs a lightweight LLM as an Oracle to validate, refine, and select the best output using schema information and parsing feedback. This division of labor lets the SLM focus on fast generation and the Oracle on correctness, creating a scalable architecture that balances efficiency and accuracy. This design is the key novelty of our work, combining the complementary strengths of SLMs and LLMs in a modular, resource-conscious framework for real-world security analytics.

We conduct extensive experiments spanning baseline performance, prompting strategies, fine-tuning, and architectural design. Our evaluation covers nine models: five SLMs (Gemma-3-1B-IT, Gemma-3-4B-IT, Phi-4-Mini-Instruct,Qwen-2.5-7B-Instruct-1M, and DeepSeek Coder 6.7B Instruct) and four LLMs (GPT-5, GPT-4o, Gemini 2.0 Flash, and Phi-4), using 230 NLQ-KQL pairs from Microsoft's NL2KQL Defender Evaluation Dataset [1] and 83 additional queries from Sentinel-Queries [35] for generalization. LLMs achieve high syntactic accuracy (above 0.90) but show variable semantic correctness, while SLMs perform poorly in zero-shot settings, with semantic scores near 0 and table/filter scores below 0.1. Targeted prompting and LoRA fine-tuning with rationale distillation significantly improve SLM performance, especially for DeepSeek Coder 6.7B Instruct with NL2KQL. Our two-stage SLM-Oracle architecture achieves a syntax score of 0.987 and semantic score of 0.906 on Microsoft's NL2KQL Defender Evaluation dataset. We demonstrate generalizability on Microsoft Sentinel data with a syntax score of 0.964 and semantic score of 0.831, approaching LLM performance while remaining up to 10x cheaper in token cost than GPT-5. These findings show that SLMs, when combined with lightweight LLM refinement, enable accurate and cost-efficient KQL generation at scale.

Our paper makes the following main contributions:

- We present the first systematic evaluation of SLMs for NLQ-to-KQL translation.
- We adapt NL2KQL for SLMs with lightweight components, introduce error-aware prompting to boost correctness without increasing token count, and apply LoRA fine-tuning with rationale distillation to transfer teacher reasoning to SLMs.
- We propose a **two-stage architecture** where an SLM generates candidate queries and a low-cost LLM refines them.
- Our approach achieves **near-LLM syntax accuracy, strong semantic performance, and up to 10x lower cost**, while maintaining low latency and generalizing effectively to unseen schemas.

Availability. Our code is available at https://github.com/DART-Laboratory/slms-for-kql

2 Related Work

Natural Language to Query Generation Converting NLQs to structured queries has been widely studied across domains and languages. Early work [5, 20, 25] translated NLQs to SQL using rule-based methods and custom heuristics, but extending these techniques to new databases and languages demands substantial manual effort and struggles with edge cases [30]. Neural methods improve portability by casting text-to-SQL as sequence generation with integrated validation. *PICARD* [27] uses sequence-to-sequence models with modular syntactic checks to ensure query correctness and alignment with the NLQ. Supplying schema context further improves semantic fidelity [14]. In practice, these ideas combine constrained decoding for syntax with schema-aware prompting or retrieval for semantics, directly informing our KQL setting. Unlike prior systems that center on a single model, our approach uses both SLMs and LLMs. These general principles have been extended to domain-specific query languages, especially in security, where KQL is the primary language for threat hunting and log analysis. *NL2KQL* [30] translates NLQs to KQL by combining embedding-based semantic similarity with few-shot prompting in a modular, end-to-end framework. *XPert* [18] takes an incident-centric approach, retrieving similar historical tickets to recommend or compose queries using in-context learning with embedding-based retrieval. However, both systems rely heavily on LLMs, resulting in cost and latency constraints that limit real-time SOC deployment, where analysts need rapid query assistance. In contrast, our approach uses an SLM for generation, avoiding exclusive dependence on LLMs.

Security-Specific LLMs LLMs are increasingly applied across SOC tasks and offensive evaluations, but their effectiveness hinges on task-specific scaffolding and verification. *VulDetect* [23] fine-tunes GPT-style models for anomaly detection in logs using a transformer-based framework. *PentestGPT* [8] builds an LLM penetration-testing assistant with modular task support, reporting bounded gains on real targets, highlighting the importance of structured tooling around the model. Although LLMs offer strong code generation capabilities, their cost and latency often make real-time deployment impractical [7], pushing real-time operations toward SLMs.

SLMs Adoption SLMs provide attractive latency and cost profiles for SOC settings but require augmentation to match LLM quality. SuperICL shows that small, locally fine-tuned models can act as plug-ins to larger LLMs, structuring context while the larger model executes—a useful division of labor for SOC workflows [32]. Parameter-efficient fine-tuning further narrows this gap while preserving efficiency; methods like LoRA and QLoRA enable targeted

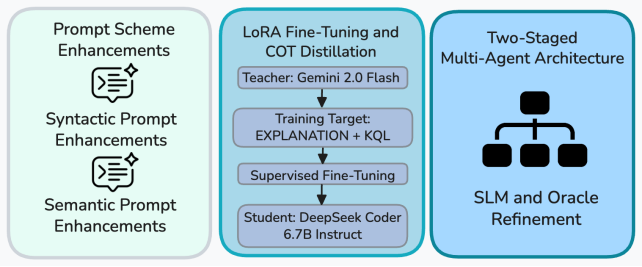

Figure 1: Orthogonal Enhancement Knobs: Prompting Scheme Enhancements Fine-Tuning with LoRA, and Multi-SLM Architecture with Oracle Refinement

adaptation on modest hardware [9, 15, 21]. Complementary techniques such as schema/value retrieval, few-shot selection, and best-of-N with an external judge improve correctness without enlarging model footprint [22, 31]. In sum, SLMs are viable for SOC use when paired with retrieval, schema hints, and parameter-efficient tuning.

3 Methodology

3.1 Problem Statement

We seek to formally define the problem space of SLM KQL code generation, which guides the design of our evaluation framework. A NLQ is defined as a request that a user wishes to be answered in terms of KQL. The set $\mathcal{T}$ represents the set of all possible NLQs. A schema is defined as the columns that are associated with each single table in KQL. The set $\mathcal{S}$ represents the set of all possible schema contexts that are associated with tables in KQL. Lastly, the set $\mathcal{Q}$ represents the set of all possible KQL queries.

Given an NLQ, $t \in \mathcal{T}$ and optional schema context $s \in \mathcal{S}$, the main objective is for a model to produce a syntactically and semantically valid KQL query $\hat{q} \in \mathcal{Q}$. We study a translator configured by three orthogonal enhancement knobs: **P** (Prompting Scheme Enhancements), **M** (Multi-SLM Architecture with Oracle Refinement), and **F** (Fine-Tuning with LoRA). Each configuration induces a mapping:

$$F_{\mathcal{L},\mathrm{P},\mathrm{M},\mathrm{F}} : (t, s) \rightarrow \hat{q}$$

for a base model $\mathcal{L}$, and we evaluate $\hat{q}$ with syntax validity, semantic equivalence, table and filter metrics, latency, and cost. Our experiments vary these knobs to quantify the marginal contribution of each knob while keeping the base model and datasets fixed.

3.2 Zero-Shot Prompting

The Zero-Shot prompting strategy aims to interact with the model without containing any relevant examples or demonstrations [26]. Zero-Shot prompting can be an extremely powerful technique, as it typically utilizes less tokens than other prompting strategies and relies mostly on the inference capabilities of the language model itself. Furthermore, instruction tuning has been shown to improve Zero-Shot learning capabilities [24].

We begin by testing with Zero-Shot prompting strategies. Zero-Shot prompting strategies allow us to establish a baseline performance for how SLMs perform in developing KQL queries without any supplemental information. Furthermore, it allows us to understand how powerful the model is in generating relevant KQL queries as a standalone system. The first prompting strategy is outlined in Figure 4 in Appendix A.

In this first prompting strategy, we set the context for the SLM to understand that they are to generate a KQL query that meets the needs of an NLQ. We then provide the NLQ and allow the SLM to think about what may constitute as a valid KQL query with respect to the NLQ provided. This allows us to establish a baseline of how SLMs perform with respect to generating KQL queries as a standalone system.

3.3 NL2KQL-Inspired Prompting

In addition to Zero-Shot prompting, we re-implement NL2KQL [30], the state-of-the-art system for KQL query generation. NL2KQL consists of five main components: Semantic Data Catalog, Schema Refiner, Few-Shot Selector, Prompt Builder, and Query Refiner. For detailed descriptions of these components, we refer the reader to the original paper [30]. Below, we describe the changes we made in our recreation.

3.3.1 Semantic Data Catalog. We recreate the Semantic Data Catalog by generating table and value embeddings from the Microsoft Defender schema. While NL2KQL relies on the text-embedding-ada-002 embedding model to generate embeddings, we use Google's `text-embedding-004` model to construct both the Table Embedding Store and Value Embedding Store. Google's text-embedding-004 model is lightweight, and provides embeddings at a cost of $0.025 per one million input tokens, compared to $0.10 per one million input tokens for `text-embedding-ada-002` embedding model.

3.3.2 Schema Refiner. Consistent with [30], the Schema Refiner retrieves the top-t relevant tables and associated columns using cosine similarity between the NLQ and the embeddings. As in the original, $t = 9$ and $v_n = 5$. We follow the same procedure but base our embeddings on the recreated Semantic Data Catalog.

3.3.3 Few-Shot Selector. The original NL2KQL constructs a Few-Shot Synthetic Database (FSDB) from synthetic NLQ-KQL pairs sampled from Microsoft Defender tables and categorized into five themes (Explore, Expansion, Detect, Remediate, Report). While it discards invalid primary and secondary queries during generation, we generate all pairs first and then assess syntactic and semantic correctness, improving overall efficiency and resource use. We use Google Gemini 2.0 Flash to build the FSDB, which enables fast generation and produces mostly correct KQL queries. As in the original, the top-t tables from the Schema Refiner filter the FSDB, and the top-f examples ($f = 2$) are selected using cosine similarity to the NLQ.

3.3.4 Prompt Builder. We preserve the design of the Prompt Builder from [30], which combines detailed instructions, syntax rules, best practices, and few-shot examples into a single prompt template.

3.3.5 Query Refiner. We implement the Query Refiner following [30], which uses Microsoft's KQL Parser [2] to verify syntactic and semantic correctness, detect undefined identifiers, repair aggregate functions, fix parentheses, and add missing operators. We retain the embedding-based replacement strategy with a cosine similarity threshold of 0.9.

3.3.6 Alternative Prompting Strategies. Beyond faithfully reimplementing NL2KQL [30], we introduce prompting strategies that supply models with targeted tips on common errors. The first focuses on frequent syntactic issues identified via the KQL Parser. The second adds guidance for semantic errors such as mismatched column types and invalid table references. We compare both strategies and adopt the one with the highest correctness. Due to SLMs' limited inference capabilities, concise, model-specific tips help them generate more accurate KQL compared to supplying the full syntax and semantic rules. Our reimplementation substitutes Google's `text-embedding-004` for embeddings, uses Google Gemini 2.0 Flash and Microsoft Phi-4 for query generation, and shifts correctness filtering to post-generation. We further extend prompting with error-aware instructions to assess whether targeted guidance improves KQL generation.

3.4 LoRA Fine-Tuning with LLM Distillation

To further assess the quality of KQL queries that are generated from SLMs, we also test LoRA, a fine-tuning method that freezes pre-trained model weights from language models, and applies decompositional matrices to transformers layers within the LLM in order to alter a subset of model weights [15]. The goal of this method is to train SLMs on NLQ-KQL pairs so that SLMs can be further improved in producing quality responses. LoRA can be mathematically defined as follows:

Given a weight matrix, $\mathcal{W}$, we decompose the weight matrix into two smaller matrices such that:

$$\mathcal{W}' = \mathcal{W} + \Delta W = \mathcal{W} + AB$$

Where $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, and both are low-rank matrices. During training time, the values of the original weight matrix $\mathcal{W}$ remain stable while ΔW (AB) is updated. Altering low-rank matrices instead of the entire weight matrix reduces the memory requirements, gradient storage, and optimizer states needed to perform fine-tuning [21]. To train the DeepSeek Coder 6.7B Instruct, we use a synthesized dataset created through LLM Knowledge Distillation.

LLM Knowledge Distillation is a transfer learning technique in Machine Learning that is used to relay the inference and reasoning capabilities of larger, proprietary LLMs to smaller SLMs. This allows the SLMs to preserve its nature of greater efficiency, deployment feasibility, and faster inference capabilities while also receiving the knowledge of LLMs in its own reasoning processes [33].

Using the same process that was used to create a FSDB when reimplementing NL2KQL, we create a synthetic dataset of 1,000 NLQ-KQL pairs that are verified to be syntactically and semantically correct using the KQL Parser. The teacher model that is used to create the synthetic dataset of NLQ-KQL pairs is Gemini 2.0 Flash. We choose Gemini 2.0 Flash as a teacher model due to lower input/output token costs while also showing promise in generating syntactically and semantically correct KQL queries. To expose the student to reasoning signals, we augment each NLQ–KQL pair with a short chain-of-thought (CoT) explanation generated by the teacher (Gemini 2.0 Flash). During supervised fine-tuning, the target sequence is a two-part output: the explanation followed by the final KQL. At inference, we report results for both *reason-then-answer* decoding (model emits explanation and KQL) and *answer-only* decoding (model emits KQL directly). This rationale-augmented setup aims to distill not only the teacher's answers but also its transformation process, improving robustness and compositional correctness without increasing model size. Then using the LoRA Fine-Tuning technique, we fine-tune a student model, DeepSeek Coder 6.7B Instruct, using supervised fine-tuning. Of the 1,000 NLQ-KQL pairs generated, 800 NLQ-KQL pairs are used for fine-tuning the SLM and 200 of the NLQ-KQL pairs are used as a validation set.

Oracle Prompting Methods

Oracle for Retrieval and General Refinement:
Given a Natural Language Query and a list of KQL queries, determine which of the following KQL queries is most syntactically and semantically correct:

Natural Language Query:
Responses:

Make changes as necessary to refine the best KQL query, and ensure that each column in the KQL query belongs to its respective table(s). Return the correct answer without explanation.

Oracle for Retrieval and Schema Context:
Given a Natural Language Query and a list of KQL queries, determine which of the following KQL queries is most syntactically and semantically correct:

Natural Language Query:
Responses:

Make changes as necessary to refine the best KQL query, and ensure that each column in the KQL query belongs to its respective table(s). You may use the following tables and columns:

{SCHEMA}

Return the correct answer without explanation.

Figure 2: Oracle prompting templates used to guide refinement of model-generated KQL queries. The first oracle uses retrieval and refinement only, while the second incorporates schema context.

3.5 Two-Staged Architecture

To improve KQL generation quality from SLMs, we adopt a Two-Staged Architecture using LangChain to route queries through a single DeepSeek Coder 6.7B Instruct instance with temperature set to 1. Using SLMs for initial generation reduces costs compared to LLMs, which are more expensive for large inputs. The SLM outputs are then passed to an Oracle model, Gemini 2.0 Flash, which selects the best syntactic and semantic response while keeping token usage low. The Oracle further refines the chosen KQL to maximize correctness. This design builds on NL2KQL but adapts it for SLMs. We retain the Semantic Data Catalog, Schema Refiner, and Prompt Builder, but replace the Query Refiner with the Oracle, which revises the KQL to ensure both syntactic and semantic validity.

As described in Section 3.3.2, the Schema Refiner retrieves the top t relevant tables based on cosine similarity. In our modified pipeline, we select top 5 instead of top 9 values and omit column values to reduce token count and prevent SLM hallucinations. Table 10 outlines the effects of varying the number of tables provided on overall metrics. Using the few-shot examples selected as described in Section 3.3.3, we feed these to a DeepSeek Coder 6.7B Instruct

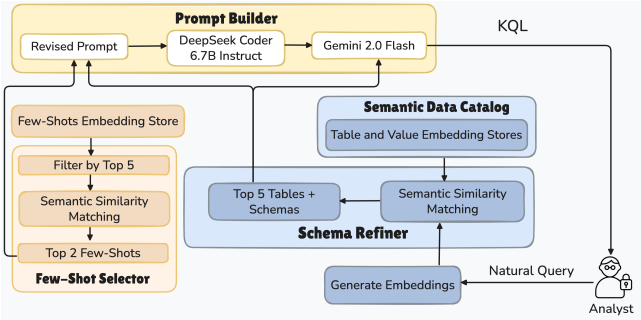

Figure 3: Two-Stage Architecture with Oracle Refinement. The NLQ is embedded to retrieve the Top 5 relevant tables, which guide the selection of Top 2 few-shot examples. These are processed by DeepSeek Coder 6.7B Instruct, and the outputs are refined by the Oracle model.

instance. We then replace the Query Refiner (Section 3.3.5) with an oracle LLM as it allows for a more automated process, and allows for different types of errors to be fixed. A full diagram of our approach can be seen in Figure 3.

The oracle operates in two modes. In *refinement-only*, it selects and edits the best SLM candidate using the judge model's internal knowledge. In *schema-aware refinement*, it performs the same process but with explicit schema context. We evaluate both modes using two prompts (Figure 2). The oracle is instantiated as an LLM-as-a-Judge, leveraging evidence that judge models align well with human preferences while exhibiting recognizable biases [34].

4 Evaluation

We evaluate the effectiveness of our proposed design through a series of experiments conducted on a machine with an AMD EPYC 9534 64-Core Processor, an NVIDIA H100 NVL GPU, and Ubuntu 22.04.5 LTS. We address seven research questions (RQs) as follows. For all RQs, we access and query SLMs using the Huggingface Python package [16], which avoids input/output token rate limits. For RQ5 and RQ6, we query Google Gemini 2.0 Flash via the Google GenAI package [11]. Our re-implemented NL2KQL pipeline (Section 3.3) is evaluated using Gemini 2.0 Flash, Phi-4, and GPT-4o.

- **RQ1**: What is the baseline KQL reasoning ability of LLMs and SLMs without enhancements?
- **RQ2**: How well does NL2KQL perform with SLMs?
- **RQ3**: How do different prompting schemes affect NL2KQL with SLMs?
- **RQ4**: How effective is LoRA fine-tuning for SLM-based KQL generation?
- **RQ5**: Can a two-stage architecture with Oracle refinement match or surpass LLM performance at lower cost?
- **RQ6**: How well does the two-stage architecture generalize to unseen schemas and tables?
- **RQ7**: What are the hyperparameter search results for different components of the two-stage architecture?

Due to space constraints, we provide results for RQ6 and RQ7 in the Appendix C.1 and Appendix D.

Model	Input Cost	Output Cost
OpenAI GPT-5	$1.25	$10.00
OpenAI GPT-4o	$2.50	$10.00
Google Gemini 2.0 Flash	$0.10	$0.40
Microsoft Phi-4	$0.075	$0.30
Qwen-2.5-7B-Instruct-1M	$0.15	$0.15
Microsoft Phi-4-Mini	$0.15	$0.15
DeepSeek Coder 6.7B Instruct	$0.15	$0.15
Gemma-3-4B-IT	$0.15	$0.15
Gemma-3-1B-IT	$0.10	$0.10

Table 1: Costs per 1 million tokens of input and 1 million tokens of output from LLMs and SLMs

Datasets RQ1-RQ5 are evaluated using Microsoft's NL2KQL Defender Evaluation Dataset [1], which contains 230 NLQ-KQL pairs referencing all tables in the NL2KQL Defender Schema. Instead of feeding NLQs directly to the models, we embed them in different prompting strategies and compare the generated queries against the baseline pairs to measure syntactic, semantic, and structural similarity. For RQ6, we obtained 83 NLQ-KQL pairs from the Microsoft Defender sections of Sentinel-Queries [35], a publicly available GitHub repository. We used all available Defender queries from this repository and excluded queries designed for other data sources.

Evaluation Metrics In order to evaluate these research questions and assess the effectiveness of our research methods, we first define the following evaluation metrics. These evaluation metrics have been previously defined in NL2KQL [30], and we adapt them as part of our evaluation process as well.

Syntax Score. evaluates whether an LLM-generated KQL query $\hat{q}$ is syntactically correct: $\text{Syntax}(\hat{q}) = 1$ if $\hat{q}$ is syntactically correct, and 0 otherwise.

Semantic Score. evaluates whether $\hat{q}$ is semantically correct within the schema s: $\text{Semantic}(\hat{q}) = 1$ if $\hat{q}$ is semantically correct, and 0 otherwise.

Table Score:. Evaluates the proportion of tables referenced in the $\hat{q}$ query $T(\hat{q})$ which are also references in q, but only if $T(q)$ is a subset of $T(\hat{q})$ otherwise, it's zero.

$$Table(q, \hat{q}) = \begin{cases} \frac{|T(q) \cap T(\hat{q})|}{|T(\hat{q})|}, & \text{if } T(q) \subseteq T(\hat{q}). \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

Filter Column Score:. Evaluates Jaccard similarity of the set of columns referenced in filters ($F_{col}(.)$) of $\hat{q}$ and q, where Jaccard of two sets is defined as $Jaccard(a, b) = |\frac{a \cap b}{a \cup b}|$

Filter Literal Score:. Evaluates Jaccard similarity of the set of literals used in filters of $\hat{q}$ and q.

In addition to adapting the metrics defined in NL2KQL, we also note two more metric measures:

Avg. Latency:. Average end-to-end model inference time per NLQ (s/query), measured from prompt assembly to final string, including network/API overhead. For the two-stage architecture, latency is the maximum per-NLQ end-to-end time across all SLMs plus the Oracle model's inference time.

Cost:. Total cost (USD) to process 230 NLQs from the NL2KQL Defender Evaluation Dataset, accounting for all input and output

tokens per model. SLMs offer several advantages, including enhanced data security and lower response latency, but cost efficiency is one of their most significant benefits. Like LLMs, SLM costs are determined by input and output token counts, with per-million-token pricing varying by provider. For this analysis, we report prices based on Cloudflare's Workers AI pricing, as highlighted in [3]. Table 1 presents the cost per 1M tokens for multiple LLMs and SLMs in USD. The cost gap is substantial: SLMs can offer up to 7.5x savings on input tokens alone compared to LLMs. When used effectively for code generation, SLMs can deliver significant performance at a fraction of traditional LLM costs.

Hyperparameters To train SLMs using Parameter-Efficient Fine-Tuning (PEFT) and LoRA, we keep the following hyperparameters stable for DeepSeek Coder 6.7B Instruct in RQ4: one training epoch, batch size of 5 per device, learning rate of 0.0002, weight decay of 0.001, maximum gradient norm of 0.3, and warmup ratio of 0.03. The optimizer used is `paged_adamw_32bit`. For supervised fine-tuning, we vary the LoRA parameters and select the combination minimizing validation loss. Specifically, we test alpha (α) values of 2, 4, and 8; rank (r) values of 1, 2, and 4; and dropout values of 0.1 and 0.2.

Embedding Model An embedding model is an ML model that can be used to convert text into a numerical representation so that it can be conceptually understood by LLMs or SLMs. The primary embedding model used in the following RQs is Google's text-embedding-004 model. The text-embedding-004 provides embedding queries for $0.025 per 1 million input tokens, and $0.02 per 1 million output tokens [12]. Furthermore, text-embedding-004 is a lightweight, simplistic embedding model that has performed well across multiple numerous benchmarks and has strong multilingual and domain-specific capabilities.

4.1 RQ1: Baseline LLM and SLM Performance

In this research question, we determine the baseline KQL reasoning capability of LLMs and SLMs without any enhancements. This helps to understand how LLMs and SLMs independently perform in generating KQL queries. We perform experiments across four different LLMs: OpenAI's GPT-5 and GPT-4o, Google Gemini 2.0 Flash, and Microsoft Phi-4, and five different SLMs: Google Gemma-3-1B-IT, Google Gemma-3-4B-IT, DeepSeek Coder 6.7B Instruct, Microsoft Phi-4-Mini-Instruct, and Qwen-2.5-7B-Instruct-1M. In these initial experiments, both LLMs and SLMs are not fed any contextual information needed to create the KQL query through the prompts. Rather, we exclusively assess their inference abilities to produce KQL queries from a simple NLQ. We use the prompt as outlined in the Zero-Shot Prompting Strategy in Figure 4. These results are shown in Table 2.

Without schema enhancements, LLMs generate syntactically correct KQL queries but struggle with semantic accuracy. OpenAI's GPT-5, GPT-4o and Google Gemini 2.0 Flash outperform Microsoft Phi-4 across all metrics, identifying relevant tables more effectively and achieving lower average latency. In contrast, SLMs perform poorly on their own: across all five tested, table score and $\text{Filter}_{\text{col}}$ remain below 0.1. While SLMs can produce syntactically valid KQL, they are largely unable to generate semantically correct queries.

```
EmailEvents
| where Timestamp between(datetime(" 2022-10-05T20:54:33Z") ..
    datetime("2022-10-05T21:05:12Z"))
| where ThreatTypes has "Phish"
| where EmailActionPolicy != "Anti-phishing user impersonation"
```

Listing 2: The KQL query above gives a list of phishing attempts over email that did not use identity immitation between 20:54.33 and 21:05.12 on 2022-10-05.

This suggests that even though SLMs have the capacity to learn the structure of KQL queries, they are not independently effective in producing useful KQL queries that can be used by security analysts. These trends are due to either SLM hallucinations of proper table names, hallucinations of column names, or simply due to referring to columns that do not belong within the referenced table's schema.

4.2 RQ2: SLMs with NL2KQL

In this research question, we determine how LLMs and SLMs perform within the NL2KQL system. We reimplement NL2KQL's architecture with some substitutions to develop a system that can be used to generate KQL queries, and first test this on the four before-mentioned LLMs. NL2KQL selects a certain subset of tables and their respective schema depending on the embedding of the natural language query, and its similarity to queries stored in a few-shot synthetic database, a database that contains realistic, example NLQ-KQL pairs. Furthermore, we test the NL2KQL configuration on the five before-mentioned SLMs. The full prompt used in this system can be referenced within the original NL2KQL paper [30]. These results are shown in Table 2.

From the initial results shown from this table, NL2KQL when partnered with SLMs improves nearly all measured metrics when compared to the initial Zero-Shot approach. Likewise, LLMs show improvement especially in developing semantically correct queries when partnered with NL2KQL. Although SLMs can produce syntactically correct KQL queries in the Zero-Shot configuration, the model struggles to identify the correct tables to use given an NLQ. However, when NL2KQL is utilized to augment the SLM's knowledge, the SLMs are able to identify proper tables and columns that are needed to not only produce a syntactically correct query but a semantically correct query as well. However each model does incur a greater cost as opposed to utilizing the Zero-Shot approach, and also incurs more costs with respect to input and output tokens. Furthermore, the average latency per query when utilizing the NL2KQL approach is higher compared to the Zero-Shot approach; this is likely due to the higher number of tokens that are given to the model at inference time. We leave the NL2KQL main Avg. Latency and Total Cost for NL2KQL, as these values were not provided in [30].

Because the DeepSeek Coder 6.7B Instruct model in the NL2KQL approach performs well across all measured metrics while providing an efficient cost, we choose to build on the DeepSeek Coder 6.7B Instruct model while using the NL2KQL configuration in order to increase syntactic and semantic scores of SLM-generated KQL queries.

Model and Configuration		Syntax	Semantic	Table	$\text{Filter}_{\text{col}}$	$\text{Filter}_{\text{lit}}$	Latency	Avg. Cost
GPT-5	Zero-Shot	0.9	0.7	0.7	0.404	0.52	35.711	$0.267
	NL2KQL	0.93	0.861	0.283	0.114	0.377	53.67	$2.018
GPT-4o	Zero-Shot	0.970	0.274	0.452	0.289	0.542	1.033	$0.136
	NL2KQL	0.961	0.878	0.696	0.385	0.549	10.008	$2.998
Gemini 2.0 Flash	Zero-Shot	0.973	0.282	0.383	0.266	0.540	1.095	$0.009
	NL2KQL	0.883	0.812	0.716	0.487	0.527	2.385	$0.107
Microsoft Phi-4	Zero-Shot	0.845	0.029	0.126	0.061	0.407	2.126	$0.012
	NL2KQL	0.694	0.516	0.604	0.342	0.424	11.733	$0.127
Microsoft Phi-4-Mini-Instruct	Zero-Shot	0.623	0.007	0.044	0.026	0.389	1.232	$0.004
	NL2KQL	0.664	0.279	0.510	0.332	0.420	5.398	$0.127
Gemma-3-1B-IT	Zero-Shot	0.443	0.007	0.0	0.0	0.294	1.373	$0.005
	NL2KQL	0.771	0.332	0.321	0.220	0.253	4.286	$0.084
Gemma-3-4B-IT	Zero-Shot	0.741	0.0	0.004	0.0	0.373	1.793	$0.008
	NL2KQL	0.803	0.446	0.629	0.317	0.495	6.859	$0.127
Qwen2.5-7B-Instruct-1M	Zero-Shot	0.826	0.003	0.026	0.009	0.479	0.947	$0.005
	NL2KQL	0.641	0.413	0.419	0.31	0.401	6.258	$0.126
DeepSeek Coder 6.7B Instruct	Zero-Shot	0.81	0.027	0.065	0.022	0.45	3.328	$0.009
	NL2KQL	0.879	0.779	0.757	0.52	0.51	9.455	$0.056

Table 2: Evaluation of LLM and SLM prompting configurations (including NL2KQL) using syntax, semantic, table score, filtering accuracy, and efficiency metrics. Latency is the average time per NLQ (s/query). Cost is the total USD to run all 230 queries, based on token prices per million.

```
EmailEvents
| where Timestamp between (datetime(2022-10-05 20:54:33) ..
    datetime(2022-10-05 21:05:12))
| where has_any(ThreatTypes, "Phish", "Phishing")
| project Timestamp, NetworkMessageId, SenderMailFromAddress,
    RecipientEmailAddress, Subject, ThreatTypes,
    DetectionMethods
```

Listing 3: This KQL query was generated by DeepSeek Coder 6.7B Instruct with the NL2KQL configuration for the same NLQ as in Figure 2. Although similar to the ground truth, the generated query is syntactically and semantically incorrect due to incorrect usage of the KQL keyword: has_any.

4.3 RQ3: Different Prompt Schemes

In this research question, we seek to understand how different prompting schemes affect NL2KQL with SLMs. In order to revise the prompt to increase the syntactic and semantic score, it is important to understand the syntactic and semantic errors that are generated from using the DeepSeek Coder 6.7B Instruct and the NL2KQL configuration. For this reason, we employ Microsoft's KQL code parser [2] to first determine the types of syntactic errors that are generated from this scenario. Table 4 in Appendix B lists some of the most common syntactic errors returned from the KQL queries.

From this table, the most frequent syntax errors were "Unexpected:'" and "The incomplete fragment is unexpected.", followed by missing delimiters such as "Expected: ;" and "Expected:) ". These issues commonly arise when the model mixes markdown fragments, emits partial SQL-like text, or produces malformed parentheses in operators such as `between`. By enforcing a single KQL output, constraining timestamp expressions, and requiring operators to appear only in infix form, Alternative Prompt #1 reduces these incomplete or ill-formed fragments. For this reason, we used the Alternative Prompt #1 strategy as outlined in Appendix B. Table 3 outlines the updated results from these tests.

With the revised prompting strategy, there is an increase in the syntax, semantic, and $\text{Filter}_{\text{col}}$ scores of KQL queries. After calculating these metrics, we also analyze the remaining types of syntax errors that exist after the prompting alteration. These results are outlined in Table 6. These results indicates that the number of syntax errors associated decreased. After taking this approach, we begin looking at alternative prompts that might be useful to improve the semantic score. In order to further improve the semantic score, it is imperative to understand the components where the model is unable to generate correct KQL queries. We utilize the same KQL code parser to further analyze the semantic errors that are returned within the KQL code. Table 7 outlines the list of the most common syntax errors returned from the KQL queries. From this table, some of the semantic errors that are given from the model involve improperly referenced columns. This could be due to either SLM hallucinations or due to referencing columns that do not belong to a certain table's schema. In either case, we seek to improve the semantic score by employing Alternative Prompt #2 as outlined in Appendix B. These results are shown in Table 3.

Despite specifying that multiple rules to assist the SLM in producing semantically correct queries, the SLM performance across all metrics measured does not appear to improve with this revised prompting technique. Compared to the first revised prompting technique, the DeepSeek Coder 6.7B Instruct model performs considerably worse across all metrics. This shows that even when given a proper table schema, SLMs cannot always verify independently whether KQL queries are semantically correct. Furthermore, Table 8 outlines that columns and other variables remain undefined even after specifying the SLM to follow the correct schemas.

4.4 RQ4: LoRA Fine Tuning

In this research question, we also seek to understand how LoRA Fine-Tuning performs on DeepSeek Coder 6.7B Instruct. Because

Category	Model Configuration	Syntax	Semantic	Table	$\text{Filter}_{\text{col}}$	$\text{Filter}_{\text{lit}}$	Latency	Avg. Cost
Baseline	NL2KQL	0.988	0.960	0.822	0.699	0.666	–	–
Prompting Strategies	NL2KQL + DeepSeek (Original)	0.862	0.71	0.626	0.398	0.496	9.455	$0.055
	NL2KQL + DeepSeek (Revised #1)	0.924	0.753	0.618	0.467	0.456	13.55	$0.2403
	NL2KQL + DeepSeek (Revised #2)	0.869	0.6	0.550	0.424	0.442	14.346	$0.257
Advanced Configurations	DeepSeek (Supervised LoRA Fine-Tuning)	0.911	0.726	0.564	0.452	0.475	13.90	$0.26
	DeepSeek (CoT LoRA Fine-Tuning)	0.909	0.716	0.577	0.47	0.477	13.91	$0.26
	Multi-Agent (General Refinement)	0.83	0.537	0.594	0.463	0.497	18.267	$0.266
	Multi-Agent (Schema Context)	0.987	0.906	0.659	0.537	0.562	12.315	$0.213

Table 3: Evaluation of NL2KQL and various model configurations on the Defender Dataset.

DeepSeek Coder 6.7B Instruct has shown improved promise in generating KQL queries with the help of revised prompting techniques, a full-fine tuning of model parameter weights may be excessive. As an alternative, LoRA Fine-Tuning provides an efficient method of fine-tuning a model by minimizing computational costs needed to alter model weights. We first perform a hyperparameter search that minimizes the cross-entropy validation set loss of DeepSeek Coder 6.7B Instruct. Once we have found the set of parameters that minimizes the validation set loss, we test this configuration on the NL2KQL system. As shown in Table 12, the SLM configuration when $\alpha = 8$, $r = 1$, and LoRA dropout = 0.2 produced an validation set loss of 0.0104. Because this configuration minimized the evaluation loss, we use the fine-tuned model from this configuration to insert into the NL2KQL system. The results from this configuration are shown in Table 3 under Advanced Configurations.

The fine-tuned model performs worse in producing syntactically and semantically correct KQL queries compared to the original prompting techniques. However, the LoRA Fine-Tuned versions of DeepSeek Coder 6.7B Instruct, when utilized in the NL2KQL system, are able to perform slightly better in generating semantically correct KQL queries. Furthermore, the costs of running the fine-tuned model, assuming the same input/output token costs as the DeepSeek Coder 6.7B Instruct, remain roughly the same.

It is possible that due to the nature of reasoning SLMs, supervised fine-tuning alone may not be sufficient in order to train SLMs. When fine-tuning SLMs to improve phishing email detection, researchers found that directly fine-tuning models small reasoning models such as Llama-3.2-3B-Instruct and Qwen-2.5-1.5B-Instruct on phishing emails and respective labels yielded poor results when compared to vanilla prompting strategies [21]. For this reason, we also attempt a CoT approach, and augment each NLQ–KQL pair with a short explanation generated by the teacher (Gemini 2.0 Flash) of how the KQL result was generated. This provides the SLM model the proper reasoning behind a KQL query result in addition to the actual KQL answer. As outlined in Table 11, since $\alpha = 8$, $r = 1$, and LoRA dropout = 0.2 minimized validation set loss to 0.0104, we use the fine-tuned model with these parameters. These results are outlined in Table 3 under Advanced Configurations. The CoT Fine-Tuning approach is comparable with the supervised fine-tuning approach in producing syntactically and semantically correct KQL queries. However, it does not surpass the different prompting strategies that have been attempted.

The drop in performance is likely due to limited training examples, possible inappropriate distillation strategies, or a combination of both factors. In our fine-tuning setup, only two modules within the DeepSeek model are targeted for fine-tuning. A further analysis of which modules should be fine-tuned could affect the overall results for whether the fine-tuned models produce better results or not.

4.5 RQ5: Two-Staged Architecture

Building on the evaluation from **RQ3**, we propose a two-Staged Architecture that combines components of the NL2KQL architecture, leverages an SLMs instance of DeepSeek Coder 6.7B Instruct, and uses an oracle LLM, Google Gemini 2.0 Flash, to verify that KQL queries are syntactically and semantically correct. We introduce the SLMs instance and set its temperature equal to 1. While SLMs can perform well at producing syntactically correct queries, leveraging an oracle LLM model to process the outputs can ensure that KQL queries that are produced are refined to produce more semantically correct queries. We choose Google Gemini 2.0 Flash due to its relatively low cost for input tokens and output tokens while also yielding syntactically and semantically correct KQL queries.

This solution queries DeepSeek Coder 6.7B Instruct, collects the result once they have been outputted by the SLM, and allow an Oracle model to refine the best query as much as possible. To test the efficacy of the two-staged architecture as a whole, we leveraged different prompting techniques for the Oracle model, as outlined in 2. This allows us to understand how useful an oracle refinement is with respect to the SLM that generate KQL queries. In the first oracle prompting technique, the oracle model chooses the best response and regenerates a proper KQL query if the best response is not syntactically or semantically correct according to the Oracle model itself. In the second oracle prompting technique, the oracle model chooses the best response and regenerates a proper KQL query with additional schema context provided from the Schema Refiner in the NL2KQL architecture. Furthermore, we test the NL2KQL configuration with respect to the best prompting strategy and assess how the system performs with respect to the Oracle methods as well. The results from these varying Oracle prompts is shown in Table 3.

From the results noted, the revised prompting strategy performs better than the original NL2KQL prompting strategy with respect to generating syntactically and semantically correct queries. Furthermore, of the two solution strategies, the two-staged architecture solution with schema context performs the best in terms of generating syntactic and semantically correct KQL queries. These metrics beat the NL2KQL Configuration that uses Gemini 2.0 Flash, while costing over ten times cheaper to run through the entire evaluation set. However, it does not beat the NL2KQL (reported) results in

terms of syntax and semantic score. When the two-staged architecture solution is told to simply pick the best solution provided by the LLMs, the two-staged architecture solution produces slightly better syntactically correct queries than NL2KQL and the revised prompting solution. However, when we allow the oracle model to refine the KQL queries generated by the SLMs, the two-staged architecture solution produces near perfect syntactically correct queries but mediocre semantically correct queries.

5 Discussion & Limitations

Practical Implications for SOC Deployment. Our results indicate that SLMs can support day-to-day NLQ→KQL authoring when paired with lightweight scaffolding. Zero-shot SLMs provide fast, low-cost drafts but require schema context to avoid table and column hallucinations. Few-shot exemplars further reduce revision effort, and error-aware tips eliminate recurring syntactic mistakes with minimal token overhead. The two-staged architecture is operationally attractive: one DeepSeek Coder 6.7B Instruct generator keeps latency bounded, while a low-cost Oracle (Gemini 2.0 Flash) selects and refines a single query under a code-only policy and a validator. For SOCs, this suggests a practical pattern: local or private SLMs for generation, a narrow LLM for judgment and refinement, and strict validation before execution.

Generalizability to Languages. The framework extends beyond SQL to EQL[10] and SPL[29] with straightforward changes. We treat EQL indexes/index patterns and SPL index–sourcetype pairs or CIM datamodels as "tables"; build schemas from Elasticsearch mappings (typed fields, keyword variants, ECS when available) or from Splunk knowledge objects and `tstats` introspection; then retrieve top-t sources and top-v fields. Few-shot selection is unchanged; examples are serialized in native syntax (EQL `event/sequence` with `where`, `by`, `with maxspan`; SPL `search`, `where`, `stats`, or `tstats` when a datamodel applies). Prompts add guardrails: EQL uses canonical timestamp comparisons and restricts fields to chosen indexes; SPL specifies `earliest/latest` in the `search` command, prefers `tstats` for datamodel fields, and avoids `table` or `timechart` unless requested. Validators swap accordingly: EQL uses a dry-run parse plus field-existence and legal `sequence/maxspan/by` checks; SPL uses the parser with per-stage field checks and `tstats` enforcement. Metrics adapt mechanically: table scores → index or index–sourcetype scores; filter scores = Jaccard over fields/literals in `where` or `stats by`, penalizing out-of-schema fields; EQL sequences must have correct stage order. To manage tokens, we cap context to top-t sources and top-v fields and include only relevant operator cheat sheets (e.g., EQL sequence syntax; SPL `stats/tstats`).

6 Conclusion

We present the first systematic study of SLMs for NLQ-to-KQL translation, introducing prompting enhancements, LoRA fine-tuning with rationale distillation, and a two-stage SLM–Oracle architecture. Our approach achieves near-LLM accuracy at up to 15× lower cost, demonstrating SLMs as a practical, scalable alternative for enterprise security analytics.

References

[1] 2023. NL2KQL Evaluation Metrics and Benchmark. https://github.com/microsoft/NL2KQL. Accessed: 2025-09-20.

[2] 2024. Kusto Query Language (KQL). https://learn.microsoft.com/azure/data-explorer/kusto/query/ Accessed: 2025-09-20.

[3] 2025. https://blog.cloudflare.com/workers-ai-bigger-better-faster/

[4] Bushra A Alahmadi, Louise Axon, and Ivan Martinovic. 2022. 99% False Positives: A Qualitative Study of SOC Analysts' Perspectives on Security Alarms. *31st USENIX Security Symposium (USENIX Security 22)*, 2783–2800.

[5] Christopher Baik, Hosagrahar V Jagadish, and Yunyao Li. 2019. Bridging the semantic gap with SQL query logs in natural language interfaces to databases. In *IEEE 35th International Conference on Data Engineering (ICDE)*. 374–385.

[6] Stefan Bargan. 2024. Introduction to KQL for SOC analysts. https://medium.com/cyberscribers-exploring-cybersecurity/introduction-to-kql-for-soc-analysts-9308f46d4597

[7] Peter Belcak, Greg Heinrich, Shihze Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. 2025. Small Language Models are the Future of Agentic AI. https://arxiv.org/pdf/2506.02153

[8] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2024. {PentestGPT}: Evaluating and Harnessing Large Language Models for Automated Penetration Testing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 847–864.

[9] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLORA: Efficient Finetuning of Quantized LLMs. *Advances in Neural Information Processing Systems (NeurIPS)* 36 (2023), 10088–10115.

[10] Elastic. 2018. Elastic Query Language Documentation. https://www.elastic.co/docs/explore-analyze/query-filter/languages/eql Accessed: 2025-10-05.

[11] Google. 2023. google-generativeai. Python Package Index (PyPI). https://pypi.org/project/google-generativeai/ Accessed: 2025-10-06.

[12] Google. 2025. Gemini API Embeddings. https://ai.google.dev/gemini-api/docs/embeddings Accessed 2025-09-20.

[13] Mohammed Hassanin and Nour Moustafa. 2024. A Comprehensive Overview of Large Language Models (LLMs) for Cyber Defences: Opportunities and Directions. https://arxiv.org/abs/2405.14487

[14] Zijin Hong, Zheng Yuan, Hao Chen, Qinggang Zhang, Feiran Huang, and Xiao Huang. 2024. Knowledge-to-SQL: Enhancing SQL Generation with Data Expert LLM. In *Findings of the Association for Computational Linguistics (ACL)*. 10997–11008. doi:10.18653/v1/2024.findings-acl.653

[15] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. LoRA: Low-Rank Adaptation of Large Language Models. *International Conference on Learning Representations (ICLR)* 1, 2 (2022), 3.

[16] Hugging Face. 2016. Hugging Face: The AI Community Building the Future. https://huggingface.co Accessed: 2025-09-20.

[17] Liming Jiang. 2024. Detecting Scams Using Large Language Models. https://arxiv.org/abs/2402.03147

[18] Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, et al. 2024. Xpert: Empowering Incident Management with Query Recommendations via Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639081

[19] Sarah Lean. 2025. What is Kusto query language? https://dev.to/techielass/what-is-kusto-query-language-2k2n

[20] Fei Li and Hosagrahar V Jagadish. 2014. NaLIR: An Interactive Natural Language Interface for Querying Relational Databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 709–712. doi:10.1145/2588555.2594519

[21] Zijie Lin, Zikang Liu, and Hanbo Fan. 2025. Improving Phishing Email Detection Performance of Small Large Language Models. https://arxiv.org/abs/2505.00034

[22] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. SELF-REFINE: Iterative Refinement with Self-Feedback. *Advances in Neural Information Processing Systems (NeurIPS)* 36 (2023), 46534–46594.

[23] Marwan Omar and Stavros Shiaeles. 2023. VulDetect: A novel technique for detecting software vulnerabilities using Language Models. In *IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE, 105–110. doi:10.1109/csr57506.2023.10224924

[24] Adrien Payong. 2024. Review: Fine-Tuned Language Models are Zero-Shot Learners. https://blog.paperspace.com/instruction-tuning/

[25] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a Theory of Natural Language Interfaces to Databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*. 149–157. doi:10.1145/604045.604120

[26] Elvis Saravia. 2024. Prompt Engineering Guide. https://www.promptingguide.ai Accessed: 2025-10-05.

[27] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural*

Zero-Shot Prompting

You are a programmer using the Kusto Query Language with Microsoft Defender. Generate a KQL query that answers the following request:

{NLQ}

Return only the KQL code without any explanation.

Figure 4: Zero-Shot prompt used to evaluate how SLMs generate KQL queries

Language Processing. Association for Computational Linguistics. doi:10.18653/v1/2021.emnlp-main.779

[28] SigmaHQ. 2017. Main Sigma Rule Repository. GitHub. https://github.com/SigmaHQ/sigma Accessed: 2025-10-06.

[29] Splunk Inc. 2024. Splunk Enterprise Search Manual. https://docs.splunk.com/Documentation/Splunk/9.4.2/Search/Aboutthesearchlanguage Version 9.4.2. Accessed: 2025-10-06.

[30] Xinye Tang, Amir H. Abdi, Jeremias Eichelbaum, Mahan Das, Alex Klein, Nihal Irmak Pakis, William Blum, Daniel L Mace, Tanvi Raja, Namrata Padmanabhan, and et al. 2025. NL2KQL: From Natural Language to Kusto Query. https://arxiv.org/abs/2404.02933

[31] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (2022).

[32] Canwen Xu, Yichong Xu, Shuohang Wang, Yang Liu, Chenguang Zhu, and Julian McAuley. 2024. Small Models are Valuable Plug-ins for Large Language Models. In *Findings of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 283–294. doi:10.18653/v1/2024.findings-acl.18

[33] Xiaohan Xu, Chongyang Tao, Tao Shen, Reynold Cheng, Jinyang Li, Can Xu, Dacheng Tao, and Tianyi Zhou. 2024. A Survey of Knowledge Distillation of Large Language Models. https://arxiv.org/pdf/2402.13116

[34] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. *Advances in Neural Information Processing Systems (NeurIPS)* 36 (2023), 46595–46623.

[35] Matthew Zorich. 2021. Sentinel-Queries: Collection of KQL Queries. GitHub. https://github.com/reprise99/Sentinel-Queries Accessed: 2025-10-06.

Appendix

A Zero-Shot Prompt

The Zero-Shot Prompt in Figure 4 provides a brief context for the SLM to understand that the focus language is KQL, and asks the SLM to produce a KQL query based on a NLQ. No additional context is given in this prompt, as this is meant to assess the SLM's understanding of KQL at a baseline level. This prompt is related to the results outlined from RQ1.

B Alternative Prompts

We introduce two revised prompt templates targeting common errors we found using Microsoft's KQL Parser; see Figure 5. Alternative Prompt 1 enforces strict schema discipline and prevents the model from generating undefined Defender-style fields. It specifies the correct infix usage of some operators such as "between", "has_any", and "contains". Alternative Prompt 2 builds on this by requiring the model to choose a valid starting table and to reference only those columns defined in that table (or in explicitly joined tables). This complements the NL2KQL setup described in Section 3.3.

Alternative Prompting Methods

Alternative Prompt #1:
You are a cybersecurity expert generating syntactically valid KQL for Microsoft Defender/Sentinel/USX. Return only a single KQL query without any explanation.

Use the following structures to describe a timestamp: between(datetime(...) .. datetime(...)) or Timestamp < ago(7d)

Use `contains`, `has`, `has_any`, `has_all`, `in`, `in~`, and `between` operators ONLY in infix form. For example: `Column has_any ("a","b")`, `Timestamp between (datetime(...) .. datetime(...))`.

Never invent new tables, columns, or values. Use only tables/columns/data types in:

{SCHEMA}

Use only values from NLQ or:

{VALUES}

Examples:

{EXAMPLES}

{NLQ}

Alternative Prompt #2:
You are a cybersecurity expert generating syntactically valid KQL for Microsoft Defender/Sentinel/USX. Return only a single KQL query inside a `kusto` code block without any explanation.

Use ONLY the following structures to describe timestamps: `Column between (datetime(...) .. datetime(...))` or `Timestamp < ago(7d)`.

Use `contains`, `has`, `has_any`, `has_all`, `in`, `in~`, and `between` operators ONLY in infix form. For example: `Column has_any ("a","b")`, `Timestamp between (datetime(...) .. datetime(...))`.

Never invent new tables, columns, or values. Use only tables/columns/data types explicitly listed in:

{SCHEMA}

If the NLQ mentions a field not present in the schema, ignore that part of the request.

Negation must use `not(...)`. Do NOT use the "!" operator anywhere.

The query must start with a table name (no leading "|") and follow a single pipeline structure.

Use only values from the NLQ or:

{VALUES}

Examples:

{EXAMPLES}

{NLQ}

Figure 5: Alternative prompting templates used to reduce common KQL errors.

C Table References

Table 4 outputs the most common syntax errors that were associated with the DeepSeek NL2KQL Configuration (RQ2). In total, there were 427 syntax errors from the LLM-generated KQL queries across five iterations. The most prevalent error type was "Unexpected: '" , followed by "The incomplete fragment is unexpected". While the third most common error, "Expected: ;" , is not particularly useful in fixing the KQL queries due to its generic nature, the

Syntax Error	Percentage of Errors
Unexpected: ‘	28.6%
The incomplete fragment is unexpected.	24.6%
Expected: ;	13.3%
Expected:)	7.5%
Expected: (	4.9%

Table 4: List of most common syntax errors associated with the DeepSeek NL2KQL Configuration over five iterations (Total Syntax Error Count: 427)

.

Semantic Error	Percentage of Errors
Unexpected: ‘	14.2%
The incomplete fragment is unexpected.	12.2%
Expected: ;	6.62%
Expected:)	3.72%
A value of type timespan expected.	2.67%

Table 5: List of most common semantic errors associated with the DeepSeek NL2KQL Configuration over five iterations (Total Semantic Error Count: 860)

.

Syntax Error	Percentage of Errors
Expected: ;	23.3%
The incomplete fragment is unexpected.	20.0%
Missing: "	11.0%
Unexpected: \	10.5%
Missing expression	9.0%

Table 6: List of most common syntax errors associated with the revised prompting strategy. (Total Syntax Error Count: 210)

fourth error, “Expected:)” (7.5%), suggests issues with improper parenthesis matching in function calls or logical expressions.

Table 5 presents the most common semantic errors from the same configuration. With 860 total errors, the semantic errors reveal more specific issues with the model’s understanding of KQL syntax and data types. The most common semantic error, “A value of type timespan expected”, indicates difficulties in constructing proper temporal expressions. Additionally, errors related to join conditions suggest the model struggles with the proper syntax for table joins in KQL. These error patterns inform the formulation of the first prompting strategy in RQ3.

Table 6 shows the most common syntax errors that were found after the changes in prompting in the DeepSeek NL2KQL Configuration. The number of syntax errors significantly decreased with the new prompting strategy.

Table 7 outlines the most common semantic errors that were found after the changes in prompting in the DeepSeek NL2KQL Configuration. Although only the top 5 common semantic errors are highlighted here, most of semantic errors involved missing column names, and incomplete query fragments. Notable among the errors are cases where column names do not refer to known columns

Semantic Error	Percentage of Errors
Expected: ;	6.2%
Column name expected.	5.4%
The incomplete fragment is unexpected.	5.3%
A value of type string or dynamic expected.	3.5%
The name ’FileName’ does not refer to any known column, table, variable or function.	3.4%

Table 7: List of most common semantic errors associated with the revised prompting strategy. (Total Semantic Error Count: 793)

Semantic Error	Percentage of Errors
Column name expected.	8.6%
Expected: ;	7.7%
The incomplete fragment is unexpected.	5.1%
Expected: ,	4.8%
The name ’FileName’ does not refer to any known column, table, variable or function.	2.3%

Table 8: List of most common semantic errors associated with the second revised prompting strategy. (Total Semantic Error Count over five iterations: 1185)

in the specified tables (3.4%), likely due to SLM hallucinations or misguided assumptions about which columns belong to certain tables.

Table 8 outlines the most common semantic errors that were found after a second change prompting in the DeepSeek NL2KQL Configuration. With the second revised prompting technique, the number of semantic errors increase. This shows that the second revised prompting strategy did not assist in reducing the number of semantic errors detected.

C.1 RQ6: Generalization

After establishing a two-staged architecture and assessing performance on NL2KQL’s evaluation dataset, we also seek to understand how this two-staged architecture performs on other evaluation datasets. In order to test how this system performs on other schemas, we utilize a different dataset from the original evaluation dataset released from NL2KQL. Therefore, we use the Sentinel-Queries Dataset [35] for evaluation, which consists of 83 open-source GitHub KQL queries. We use the same prompting strategy utilized in the Two-Staged Architecture Solution w/Schema Context.

When given a new dataset, the two-staged architecture performs well in developing syntactically and semantically correct KQL queries. However, the architecture struggles significantly with column filtering (0.267), and especially literal filtering (0.1). This shows that although the two-staged architecture shows strong promise in producing syntactically and semantically correct KQL queries, it may produce misleading results due to incorrect table selection and poor filtering accuracy.

Configuration (w/NL2KQL)	Syntax	Semantic	Table	$Filter_{col}$	$Filter_{lit}$	Lat.	AC.
Two-Staged Architecture Solution (Schema Context)	0.964	0.831	0.429	0.267	0.1	12.37	$0.078

Table 9: Evaluation of NL2KQL and multiple SLM configurations on the Sentinel-Queries Dataset. Lat. stands for Latency; AC. stands for Average Cost

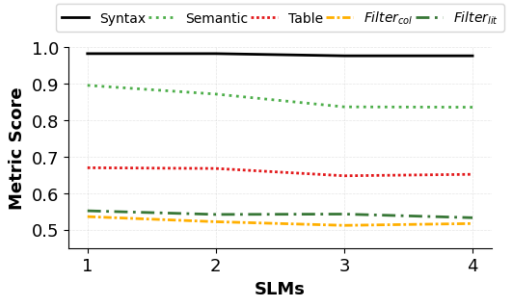

Figure 6: Number of SLMs (1-4) vs. Metric Scores

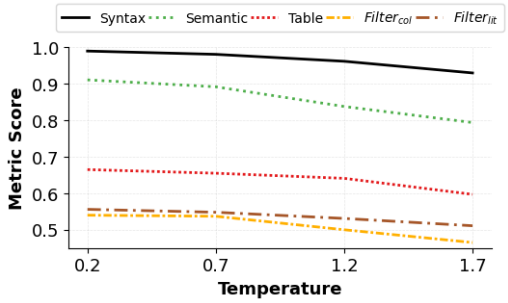

Figure 7: Temperature vs. Metric Scores

D RQ7: Hyperparameter Search Studies

In the following section, we vary different components of the two-staged architecture and assess how this affects the overall results of the system. The three main areas of testing include the **number of SLMs** that are used within the two-staged architecture system, the different **temperature values** used within the two-staged architecture system, and the **number of tables** fed into the SLM and Oracle LLM prompt.

Number of SLMs. The number of SLMs that are utilized in the two-staged architecture solution can potentially introduce a trade-off between metric scores and latency. A greater number of SLMs utilized in the system often means that the system will likely incur higher latency. However, the greater the number of SLMs that are utilized in the two-staged architecture solution, the more plausible responses are generated. In this hyperparameter search study, we determine what effect the number of SLMs utilized has on responses while keeping other components (i.e. Temperature) consistent. Figure 6 outlines the metric scores as the number of SLMs utilized in the system increases. Increasing the number of SLMs utilized in the system reduces all metric scores in the entire two-staged system. When the number of SLMs utilized in the two-staged system is one, the syntax score is 0.983, the semantic score is 0.896, the table score is 0.67, the $Filter_{col}$ score is 0.536, and the $Filter_{lit}$ score is 0.552. However, when the number of SLMs utilized in the two-staged system increases to two, the syntax score stays the same, but the semantic score is reduced to 0.872, the table score is reduced to 0.66, the $Filter_{col}$ score is reduced to 0.522, and the $Filter_{lit}$ score is reduced to 0.542. This reduction trend continues as the number of SLMs utilized in the system continue to increase as shown by the figure. Therefore, the best number of SLMs to utilize in the multi-stage system is one.

Alpha	Rank	LoRA Dropout	Validation Loss
8	1	0.2	0.010477
8	1	0.1	0.010488
8	2	0.2	0.010500
8	2	0.1	0.010526
8	4	0.2	0.010882
8	4	0.1	0.010882
4	2	0.1	0.017084
4	2	0.2	0.018105
4	1	0.2	0.018485
4	1	0.1	0.018823
4	4	0.1	0.019539
4	4	0.2	0.019622
2	2	0.1	0.026694
2	2	0.2	0.026819
2	1	0.1	0.029698
2	4	0.1	0.029962
2	4	0.2	0.030118
2	1	0.2	0.030627

Table 12: Hyperparameter Search with Validation Losses per combination (Supervised Fine-Tuning)

Different Temperature Values. We tested the two-staged architecture system under four different temperature values: 0.2, 0.7, 1.2, 1.7. These temperature values are meant to represent low, moderate, and high temperature values respectively. When configured within SLMs, low temperatures provide more deterministic code while higher temperatures introduce more randomness and creativity in token generation. In this hyperparameter search study, we study how varying the temperature of the SLMs can affect the quality of KQL queries that are produced from the proposed system. Figure 7 outlines the metric scores as the temperature utilized in the SLM within the two-staged system varies. When the temperature is set to 0.2, the syntax score is 0.99, the semantic score is 0.911, the table score is 0.665, the $Filter_{col}$ score is 0.54, and the $Filter_{lit}$ score is 0.556. When the temperature is set to 0.7, the syntax score is reduced to 0.981, the semantic score is reduced to 0.892, the table score is reduced to 0.655, the $Filter_{col}$ score is reduced to 0.537, and the $Filter_{lit}$ score is reduced to 0.548. When the temperature is set to 1.2, the syntax score is 0.962, the semantic score is 0.838, the table score is 0.641, the $Filter_{col}$ score is 0.5, and the $Filter_{lit}$ score is 0.531. Lastly, when the temperature is set to 1.7, the syntax score is 0.93, the semantic score is 0.794, the table score is 0.597, the $Filter_{col}$ score is 0.465, and the $Filter_{lit}$ score is 0.511. There does appear to be a clear relationship between temperature and performance, with all metric scores decreasing as the temperature increases.

Category	Model Configuration	Syntax	Semantic	Table	$\text{Filter}_{\text{col}}$	$\text{Filter}_{\text{lit}}$	Latency	Avg. Cost
Multi-Agent (Top t tables)	Multi-Agent (Top 1)	0.983	0.849	0.584	0.515	0.544	10.175	$0.15
	Multi-Agent (Top 3)	0.983	0.887	0.637	0.523	0.55	11.361	$0.183
	Multi-Agent (Top 5)	0.987	0.906	0.659	0.537	0.562	12.315	$0.213
	Multi-Agent (Top 7)	0.989	0.886	0.68	0.545	0.552	13.342	$0.244
	Multi-Agent (Top 9)	0.978	0.877	0.703	0.55	0.561	14.256	$0.274

Table 10: Evaluation of Multi-Agent System with varying numbers of tables.

Alpha	Rank	LoRA Dropout	Validation Loss
8	1	0.2	0.010473
8	1	0.1	0.010489
8	2	0.2	0.010507
8	2	0.1	0.010515
8	4	0.2	0.010867
8	4	0.1	0.010871
4	2	0.1	0.017208
4	2	0.2	0.018153
4	1	0.2	0.018505
4	1	0.1	0.018788
4	4	0.1	0.019508
4	4	0.2	0.019609
2	2	0.1	0.026689
2	2	0.2	0.026858
2	1	0.1	0.029772
2	4	0.1	0.029944
2	4	0.2	0.030118
2	1	0.2	0.030347

Table 11: Hyperparameter Search with Validation Losses per combination (CoT Fine-Tuning)

Different Number of Tables. We tested the two-staged architecture when it receives varying numbers of top tables ranging from 1 to 9 tables. The goal in providing fewer tables is to reduce total costs and latency in utilizing the system. However, we also seek to determine how varying the number of tables given to the SLM and Oracle LLM affects other metrics. These results are shown in Table 10.

The syntax scores remain relatively stable across varying numbers of tables supplied, ranging from 0.978 to 0.989. The semantic score generally increases as the number of tables supplied increases, reaching a peak of 0.906 with five tables before slightly declining with seven and nine tables. The table score shows a clear upward trend, increasing from 0.584 with one table to 0.703 with nine tables. However, this improvement comes at a cost: both latency and average cost increase substantially, with latency growing from 10.175 seconds (1 table) to 14.256 seconds (9 tables), and cost increasing from $0.15 to $0.274. To reach a good balance between cost, efficiency, and metrics, we choose our ideal number of tables to supply to the system as five, which achieves the highest semantic score (0.906) while maintaining reasonable latency (12.315 seconds) and cost ($0.213).

Different Alpha and Rank Values. In Table 12 and Table 11, we highlight the validation losses obtained from each combination of parameters while training DeepSeek Coder 6.7B Instruct on NLQ-KQL pairs. Table 12 highlights the validation loss in the Supervised Fine-Tuning case, while Table 11 highlights the validation loss in the CoT Fine-Tuning case.