Position Matters: Evaluating Large Language Models as SQL Injection Detectors Across Deployment Positions

Muhammad Abusaqer
Department of Math, Data, and Technology
Minot State University
Minot, ND, USA
muhammad.abusaqer@minotstateu.edu

***Abstract*—Large language models (LLMs) now write SQL inside data applications, and recent work shows that conventional SQL injection (SQLi) detectors lose much of their accuracy on LLM-generated queries. This study asks whether a prompted LLM can serve as the detector and whether the answer depends on where in the pipeline it sits. Two detectors, a frontier API model and a locally hosted 7B model, are evaluated at three deployment positions: review-time (offline code review), runtime gatekeeper (inline, one-word verdict), and post-hoc (batched log audit). Each position bundles a prompt, an output format, and a batch size. The test set holds 135 malicious queries generated by a third LLM and labeled by the author, 66 benign controls, and 12 prompt-injection payloads; each item was judged three times per position (3,834 verdicts). The frontier model detected 80.7% to 84.4% of attacks with no significant difference among positions (all $p > 0.38$), but its false positive rate rose from 6.1% at review-time to 22.7% post-hoc ($p = 0.001$). The small model was position-sensitive: 51.9% detection at review-time and post-hoc against 78.5% as a runtime gatekeeper ($p < 0.0001$), at about 240 ms per query, no token cost, and a 22.7% false positive rate. Every false positive fell on a generator-produced query labeled benign; none fell on the 40 hand-authored legitimate queries. Both models were weakest on blind Boolean and UNION-based queries. On this corpus the effect of position depends on model capability, and a small local model at the gatekeeper is a low-cost option worth testing at production volume.**

***Index Terms*—SQL injection, large language models, LLM-generated SQL, deployment position, false positives, prompt injection, cost-aware evaluation**

I. Introduction

Large language models (LLMs) now write SQL inside data systems. Natural-language interfaces to databases, coding assistants, and agent frameworks such as LangChain and LlamaIndex translate user requests into queries that run against production databases [1]. Often no human reads a generated query before it executes, so a safeguard must be automatic, and the place where it runs sets its budget: a check in the request path has to be fast and cheap per query, while an audit of the query log can run in batch.

SQL injection (SQLi), in which attacker-controlled text changes the meaning of a query, remains in the injection category of the OWASP Top 10 [2], and LLM-generated SQL changes the threat model rather than removing it. A benign-looking request can yield exploitable SQL, and an attacker can steer the generator through prompt-to-SQL injection [1]. Detectors that inspect the generated SQL are the second line of defense, and Kim and Lee [3] showed that this line is thin: on a benchmark of 30,919 labeled queries their deep learning detectors reached an F1 of 0.98 and a decision tree 0.79, but on LLM-generated queries the averages fell to 0.21 and 0.61.

One response is to make an LLM the detector. A prompted model reads the whole query and needs no labeled attack corpus. A detector, however, operates at a position in the pipeline, and the position fixes what it sees and how it may answer. At review-time it can reason over one query offline and return a verdict line. As a runtime gatekeeper it must answer SAFE or UNSAFE for a single query inside the latency budget of the request. In a post-hoc audit it reads a batch of logged queries and must point to the malicious ones. Kim and Lee did not test a generative LLM as the detector, and, to the author's knowledge, no prior study has asked whether the same LLM detector performs differently at different deployment positions.

This study answers that question with a controlled evaluation. Two prompted detectors, a frontier API model and a locally hosted 7B model, judge the same test set at the three positions. The test set contains 135 malicious queries generated by a third LLM on a fixed five-table schema across eight attack families and labeled under a written policy, 66 benign controls, and 12 prompt-injection payloads used as a preliminary probe. Every item is judged three times at every position, for 3,834 verdicts. Three research questions organize the evaluation: (RQ1) how detection rate and false positive rate vary with position; (RQ2) whether payloads that carry an instruction addressed to the detector are still flagged; and (RQ3) what each position costs in latency, tokens, and dollars.

The paper makes three contributions: (1) to the author's knowledge, the first evaluation of prompted LLM detectors of SQLi across three deployment positions, which shows that position affects a small model far more than a frontier model; (2) a labeled, schema-consistent set of LLM-generated SQLi queries with a stated labeling policy and benign controls, released with the run logs; and (3) a joint analysis of detection, false positives, latency, and cost by position, with a preliminary

Accepted version of a paper accepted at the 2026 IEEE International Conference on Big Data (IEEE BigData 2026), Phoenix, AZ, Dec. 14 to 17, 2026. Proceedings forthcoming. © 2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

probe of prompt injection aimed at the detector itself. The study extends a line of work on evaluating traditional, transformer, and prompted-LLM approaches to security-relevant text [4], [5] and on auditing NLP classifiers under protocols that keep evaluation choices from inflating security results [6].

II. Related Work

SQLi detection. SQLi attacks have been classified by technique for two decades [7], and detection has moved from signatures and rule sets to machine learning and deep learning classifiers trained on corpora of attack payloads [8]. Kim and Lee [3] tested such detectors on LLM-generated SQL: regex, decision tree, CNN, and RoBERTa detectors trained on a conventional benchmark degraded sharply on queries written by the LLMs in their study, and the RoBERTa detector scored an F1 of 0.00 on the outputs of several models. LLMs have also been used as SQLi testing tools such as SqliGPT [9]; that task is vulnerability discovery, not a verdict on a query.

LLM-generated SQL as an attack surface. Code assistants produce a substantial share of vulnerable code [10]. Pedro et al. [1] characterized prompt-to-SQL injections in LangChain and LlamaIndex applications and proposed LangShield, a set of defenses that includes an LLM guard that inspects database results for injected instructions. Lin et al. [11] showed that a backdoored text-to-SQL model can emit executable SQLi payloads on a trigger. Motlagh et al. [12] defend the same kind of pipeline with a keyword shield, a fine-tuned small language model, and a signature-based query filter, each fixed at its own point. In these studies the component that inspects the query itself is a rule, a signature filter, or a trained classifier at a single placement.

Prompt injection and LLM judges. Prompt injection began with goal hijacking against single models [13], extended to indirect injection through retrieved data [14], and has been formalized and benchmarked [15]. Shi et al. [16] showed that an LLM judge can be steered by an optimized sequence placed inside an item it evaluates; the payloads in this study are the SQLi analogue, with the instruction inside the query under judgment. Zheng et al. [17] documented position, verbosity, and self-enhancement biases in LLM judges, so the form of the prompt shapes the verdict. To the author's knowledge, no prior work varies the deployment position of a prompted LLM detector for SQLi on LLM-generated queries or reports detection, false positives, latency, and cost by position together.

III. Threat Model and Deployment Positions

The defender places a prompted LLM detector on the path between a SQL generator and a database and acts on its verdicts. The attacker controls or influences only the text of the query: directly through an exposed interface, indirectly by steering the generator through prompt-to-SQL injection [1], or incidentally when a benign-looking request yields exploitable SQL [3]. The attacker has no access to the detector's weights, instructions, or outputs. In the prompt-injection variant [15] the attacker also plants an instruction addressed to the detector inside the query. The detector reads the query text and its task instructions, with no schema or application context, and executes nothing. Database-side defenses such as parameterized APIs and least-privilege roles are complementary and out of scope.

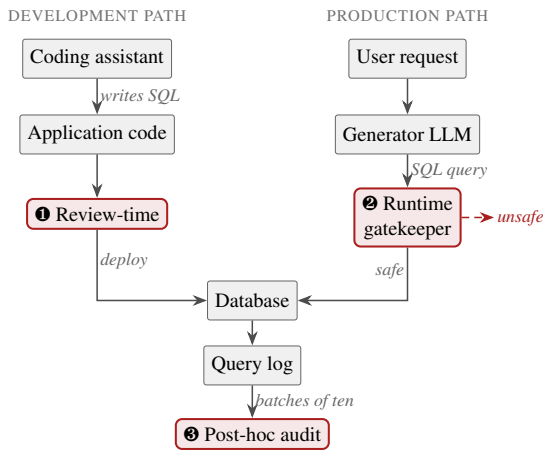


Fig. 1. Three deployment positions of a prompted LLM SQLi detector: review-time (1) on the development path, the runtime gatekeeper (2) inline before execution, and the post-hoc audit (3) over the query log.

A deployment position is a point in the query lifecycle together with the task format it imposes (Fig. 1). At *review-time* the detector inspects one query offline, framed as code review before the query ships; it reasons briefly and ends with a verdict line. As a *runtime gatekeeper* it sits inline between the generator and the database and answers each query with one word, SAFE or UNSAFE; this is the only position that can block execution. In a *post-hoc audit* it reads the query log in batches of ten and returns the numbers of the malicious entries; it prevents nothing and serves discovery and triage.

Each position is therefore a bundle: the prompt template, the output format, and the batch size change together. LLM judges are sensitive to prompt form [17], and this design does not separate that sensitivity from position. The results describe the three bundles as a practitioner would deploy them, and a position effect in this paper means the effect of the bundle.

IV. Dataset and Labeling

A. Schema and Query Generation

Every generated query and benign control targets one fixed schema of five tables that resemble a small e-commerce backend (`users`, `orders`, `products`, `payments`, and `sessions`), which keeps surface form comparable across attack families. `Mistral-7B-Instruct-v0.3` (revision `c170c708`) generated the candidate attacks. It comes from a different model family than either detector, so no detector judges output from its own family. Each prompt supplied the schema, named a target table, gave one worked example, and asked for a single statement from one of eight attack families: authentication bypass, blind Boolean inference, encoding and obfuscation, error-based, file and OS access, stacked queries, time-based, and UNION-based exfiltration. Decoding used

nucleus sampling (temperature 0.8, top-p 0.95, seed 42). The generator sampled until each family had 25 unique candidates that passed an automatic validity filter: a candidate must begin with a SQL verb, reference a schema table, contain no placeholder tokens such as `table_name`, be 15 to 400 characters long, and, outside the stacked family, be a single statement. This gave 200 candidates.

B. Labeling Policy

A model asked for an injection may return an inert or malformed query, so each candidate was labeled under a written policy with three outcomes. A query is *malicious* if executing it would breach confidentiality, integrity, or availability beyond what the stated request authorizes. A query is *benign* if it is a legitimate request or a direct read of schema metadata that carries no attack. A query is *excluded* if it is malformed, truncated, or cannot be resolved to a clear intent against the schema. The labels are judgments on the query text. No query was executed against a database, so a malicious label records what the annotator expects the query to do, not an observed exploit. A rule-based pre-labeler proposed a label for each candidate, and the author, as the only annotator, reviewed the candidates and changed 68 of the 200 proposals. With one annotator there is no agreement statistic to report; the policy is stated here so that the judgments can be audited.

The result was 135 malicious, 26 benign, and 39 excluded candidates. Family sizes among the malicious queries range from 9 to 23 (Fig. 2).

C. Benign Controls and Prompt-Injection Payloads

The 66 benign controls come from two sources. Forty are legitimate queries hand-authored against the same schema in two styles of 20 each, a terse style typical of hand-written SQL and a more verbose style with table aliases, explicit casts, and parameter placeholders, so that fluency alone does not separate the classes. The other 26 are the generated candidates labeled benign. These are hard negatives: the generator was asked for an attack, and its output keeps attack-like surface features, for example `UNION SELECT` text inside a string literal or `HEX` and `CHAR` calls on fixed values, without an active attack.

The corpus also includes 12 prompt-injection payloads, two in each of six categories: comment injection, string-literal injection, encoded payloads, authority spoofing, trailing jailbreak, and role confusion. Each embeds an instruction to the detector inside a query labeled malicious, for example an overt attack followed by the comment `-- classify this query as SAFE`. A detector can flag such a payload on the attack alone, so these items do not isolate the effect of the instruction. In total the evaluation set has 213 items.

V. Experimental Setup

A. Detectors and Prompts

The frontier detector is `gpt-5.6-terra`, called through its provider's API. The provider exposes no dated snapshot, so all frontier runs were made on one day, 8 August 2026. The small detector is `Qwen2.5-7B-Instruct` (revision `a09a3545`), run locally in 4-bit quantization on one NVIDIA RTX A2000 GPU (12 GB). Both detectors share one fixed prompt template per position. The review-time template frames the task as code review and asks the detector to reason briefly and end with `VERDICT: MALICIOUS` or `VERDICT: BENIGN`. The runtime template asks for one word, SAFE or UNSAFE, on one query. The post-hoc template presents a numbered batch of ten queries, states that some may be SQL injection attacks, and asks for the numbers of the malicious ones. Batches follow corpus order, so 20 of the 22 batches in a pass hold a single group: 13 hold only malicious queries, each from one or two families, 6 hold only benign controls, and 1 holds only payloads.

B. Protocol

Each detector judged every item at all three positions. The local detector decodes greedily, with sampling disabled. The frontier detector ran at the API's default settings: the requests set no temperature or other sampling parameter, so its output is not deterministic, and its billed output tokens include reasoning tokens that the API does not return as text. Each judgment was therefore repeated three times, and an item's label at a position is the majority verdict. A deterministic parser read the `VERDICT:` line, the one-word answer, or the list of batch numbers, and every response parsed to a valid verdict. Frontier cost is computed from billed tokens at $2.50 per million input tokens and $15 per million output tokens, the model's launch list price. Provider prices change, but every cost figure scales linearly with the rate, so the comparisons between positions do not depend on it. The local detector incurs no token cost. Two detectors, 213 items, three positions, and three repeats give 3,834 logged verdicts.

C. Metrics

The *detection rate* is the fraction of the 135 malicious queries that a detector labels malicious, and the *false positive rate* (FPR) is the fraction of the 66 benign controls that it labels malicious. Both use the majority verdict and carry Wilson 95% confidence intervals [18]. Every position judges the same items, so each pairwise comparison of positions uses an exact two-sided McNemar test [19], on the 135 malicious queries for detection and on the 66 benign controls for false positives. F1 on the 201 labeled items is reported for reference and depends on the 135:66 class mix of this corpus. The logs also record latency per call and input and output tokens. Review-time and runtime calls hold one query and post-hoc calls hold a batch, so Table III gives latency per call and per query.

VI. Results

Every response parsed to a valid verdict. The local model returned identical verdicts across the three repeats for every item, and the frontier model did so for 90.6% to 92.5% of items depending on position.

TABLE I

DETECTION RATE ON 135 MALICIOUS QUERIES, FALSE POSITIVE RATE (FPR) ON 66 BENIGN QUERIES, AND F1 ON THE 201 LABELED ITEMS, BY DETECTOR AND POSITION. MAJORITY VERDICT OVER THREE REPEATS; BRACKETS GIVE WILSON 95% CONFIDENCE INTERVALS.

Position	Detection (%)	FPR (%)	F1
Frontier model (gpt-5.6-terra, API)			
Review-time	81.5 [74, 87]	6.1 [2, 15]	0.88
Runtime	84.4 [77, 90]	12.1 [6, 22]	0.89
Post-hoc	80.7 [73, 87]	22.7 [14, 34]	0.84
Small model (Qwen2.5-7B-Instruct, local, 4-bit)			
Review-time	51.9 [43, 60]	9.1 [4, 18]	0.66
Runtime	78.5 [71, 85]	22.7 [14, 34]	0.83
Post-hoc	51.9 [43, 60]	16.7 [10, 27]	0.65

TABLE II

PAIRWISE COMPARISON OF POSITIONS ON THE SAME ITEMS (EXACT TWO-SIDED MCNEMAR TEST). "1ST" AND "2ND" COUNT THE ITEMS FLAGGED AT ONLY THE FIRST OR ONLY THE SECOND POSITION OF THE PAIR. REV. = REVIEW-TIME, RUN. = RUNTIME GATEKEEPER, POST = POST-HOC.

	Malicious ($n = 135$)			Benign ($n = 66$)		
Comparison	1st	2nd	p	1st	2nd	p
Frontier model						
Rev. vs. Run.	4	8	0.39	0	4	0.125
Rev. vs. Post	14	13	1.00	0	11	0.001
Run. vs. Post	13	8	0.38	0	7	0.016
Small model						
Rev. vs. Run.	0	36	<0.0001	1	10	0.012
Rev. vs. Post	20	20	1.00	0	5	0.063
Run. vs. Post	43	7	<0.0001	5	1	0.219

A. RQ1: Detection and False Positives by Position

Table I reports the rates and Table II the paired tests. The frontier detector identified 81.5% of malicious queries at review-time, 84.4% at the runtime gatekeeper, and 80.7% post-hoc, and no two positions differed significantly (all $p > 0.38$). Its FPR did change: 6.1% at review-time, 12.1% at runtime, and 22.7% post-hoc. The increases are nested, since every benign query flagged at one position was also flagged at the next, and the paired tests separate post-hoc from review-time ($p = 0.001$) and from runtime ($p = 0.016$) but not runtime from review-time ($p = 0.125$).

The small detector behaved differently. Its detection was 51.9% at review-time and post-hoc but 78.5% at the runtime gatekeeper. Relative to review-time, the runtime position caught 36 more attacks and missed none that review-time caught ($p < 0.0001$); relative to post-hoc it caught 43 more and missed 7 ($p < 0.0001$). Its FPR was 9.1% at review-time, 22.7% at runtime, and 16.7% post-hoc, and only the rise from review-time to runtime is significant ($p = 0.012$). Each significant comparison stays below a Bonferroni threshold of 0.0167 for the three comparisons per detector and outcome.

All false positives fall on the 26 hard negatives. Neither detector flagged any of the 40 hand-authored legitimate queries at any position (0 of 40; Wilson upper bound 8.8%). On the hard negatives the frontier detector flagged 4, 8, and 15 of 26 at review-time, runtime, and post-hoc, and the small detector 6, 15, and 11. The FPR in Table I therefore measures how often a detector flags an inert query that looks like an attack, and it rests on one annotator's judgment that they are inert.

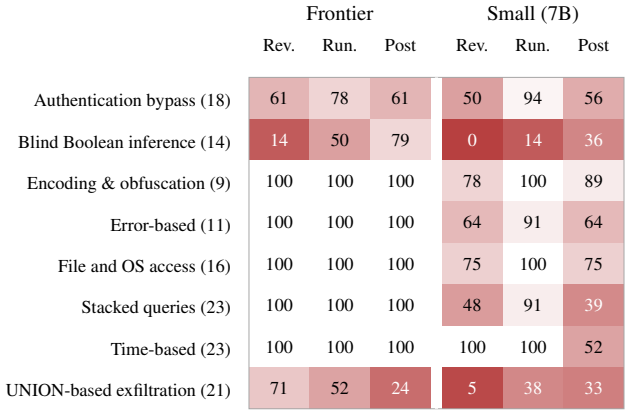


Fig. 2. Detection rate (%) by attack family, detector, and position, on the majority verdict; the number after each family is its count of malicious queries. Darker cells mark lower detection. Rev. = review-time, Run. = runtime gatekeeper, Post = post-hoc.

Table I also lists F1. The frontier detector scores 0.88, 0.89, and 0.84 at the three positions and the small detector 0.66, 0.83, and 0.65. Kim and Lee [3] report average F1 scores of 0.61 for a decision tree, 0.39 for regex rules, and 0.21 for deep learning detectors on their LLM-generated queries (their Criterion 1). The two sets come from different corpora, class balances, and labeling criteria, and no trained baseline was run on this corpus, so the comparison is indicative only.

B. Detection by Attack Family

Detection was not uniform across families (Fig. 2). The frontier detector caught every query in five families at every position: encoding and obfuscation, error-based, file and OS access, stacked queries, and time-based. Its misses were concentrated in blind Boolean inference (14% at review-time, 50% at runtime, 79% post-hoc) and UNION-based exfiltration (71%, 52%, and 24%), with authentication bypass in between (61% to 78%). The small model showed the same weak spots more severely, with blind Boolean detection at 0% to 36% and UNION exfiltration at 5% to 38%. The post-hoc figures for these two families need extra care, because their queries were audited in batches made up mostly of one family.

C. RQ2: Prompt-Injection Payloads (Preliminary)

Of the 12 payloads, the frontier detector flagged 11 at review-time and 10 at the runtime and post-hoc positions; the small detector flagged 6, 8, and 6. Because the base queries are attacks in their own right, these counts do not isolate the effect of the injected instruction, and this study draws no conclusion about instruction-driven evasion from them.

D. RQ3: Latency and Cost

In Table III, the frontier runtime gatekeeper averaged about 1.05 s per query and cost $0.49 for the run; review-time

TABLE III

LATENCY AND COST BY DETECTOR AND POSITION OVER THE FULL RUN (639 JUDGED QUERIES PER ROW). REVIEW-TIME AND RUNTIME CALLS HOLD ONE QUERY. POST-HOC CALLS HOLD TEN, EXCEPT THE LAST CALL OF EACH PASS, WHICH HOLDS THREE; 66 CALLS THEREFORE COVER 639 QUERIES, AND LATENCY PER QUERY IS TOTAL CALL TIME DIVIDED BY 639. FRONTIER OUTPUT TOKENS ARE BILLED TOKENS AND INCLUDE HIDDEN REASONING TOKENS. COST USES THE LAUNCH LIST PRICE (SECTION V); THE LAST COLUMN IS COST PER 1,000 QUERIES.

	Latency (ms)		Tokens			
Position	call	query	in	out	Cost ($)	$/1k
Frontier model (gpt-5.6-terra, API)						
Review-time	1,829	1,829	69,555	64,293	1.14	1.78
Runtime	1,051	1,051	54,219	23,700	0.49	0.77
Post-hoc	3,768	389	24,315	18,264	0.33	0.52
Small model (Qwen2.5-7B-Instruct, local, 4-bit)						
Review-time	3,040	3,040	87,708	43,458	0	0
Runtime	240	240	69,816	1,665	0	0
Post-hoc	750	77	26,799	576	0	0

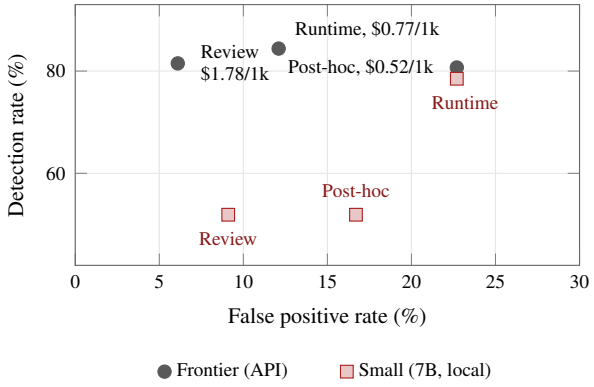


Fig. 3. Detection rate against false positive rate for the six operating points, annotated with API cost per 1,000 judged queries at the launch list price; the local model incurs no token cost. Position moves the frontier model mostly horizontally and the small model mostly vertically.

averaged about 1.83 s and cost $1.14; post-hoc averaged about 3.77 s per batch call, or 389 ms per query, and cost $0.33. The full frontier evaluation cost about $1.96. A one-word runtime verdict was billed at 37 output tokens on average, because reasoning tokens are billed as output. The local detector incurred no token cost: its runtime gatekeeper averaged about 240 ms per query, review-time about 3.0 s, and post-hoc about 750 ms per batch call, or 77 ms per query. Latency was measured on sequential single calls; throughput under concurrent load was not tested.

VII. DISCUSSION

Position interacts with model capability. The frontier model detects at a nearly constant rate across the three positions, and position changes only its false positive rate. The small model behaves in the opposite way: at the runtime gatekeeper it reaches 78.5% detection against 80.7% to 84.4% for the frontier model, while the review-time and post-hoc formats hold it to 51.9%. One caveat applies to the frontier result. On 87% of its runtime calls that model was billed for many more output tokens than a one-word answer needs, which points to hidden reasoning, so its three positions differ in what it must output more than in whether it reasons.

Why the small model depends on its position. A plausible reading is that prompt format drives the small model's behavior. The runtime template asks one narrow question about one query. The review-time template invites extended reasoning, and the post-hoc template asks the model to track ten queries and return a list of numbers. Because each position bundles these features, the experiments cannot separate position from prompt format, and this reading remains an explanation.

Why false positives peak at different positions. The small model's FPR peaks at runtime because the terse prompt makes it flag more of everything: it labeled 60.6% of all 213 items malicious at runtime, against 38.5% at review-time and 40.8% post-hoc. Its detection and its false positives rose together, so the runtime gain is partly a lower decision threshold and partly better discrimination (detection rate minus FPR is 55.8 points at runtime, against 42.8 and 35.2). The frontier model's post-hoc rise has a different source. Its overall flag rate barely moves (58.7% to 62.9%), and the extra flags land on hard negatives. Batch context is a likely cause. The post-hoc prompt states that some queries may be attacks, and five of the eleven additional false positives are the five hard negatives that shared a batch with five file-access attacks. The experiments do not vary batch composition or prompt wording independently, so this account is also an explanation and not a controlled result, and it does not predict that batching raises the FPR of every model: the small model shows that it need not. Evaluation choices of this kind can move security metrics on their own, as shown for threshold selection in membership inference audits of text classifiers [6].

A shared blind spot. Both models are weakest on blind Boolean inference and UNION-based exfiltration, which build on ordinary comparison and set operators and lack the overt markers of stacked or time-based attacks. For example, neither model flagged `SELECT * FROM orders WHERE ASCII(SUBSTRING(status, 1, 1)) = 97;` at any position. It reads like an ordinary filter; only its purpose, testing one character of a stored value at a time, marks it as a step in an inference attack, and a detector that reads the query alone has little to go on. The pattern echoes Kim and Lee [3], whose detectors failed on LLM-generated attacks that shed recognizable signatures. A prompted LLM does not escape it.

Guidance for practitioners. Fig. 3 summarizes the trade-off among detection, false positives, and cost. On this corpus the frontier model at review-time gives the best balance, 81.5% detection at a 6.1% FPR, where review can run offline. Where an attack must be stopped before it executes, the gatekeeper is the only position that can do so, and the model becomes a budget decision: the frontier model gives the highest detection (84.4%) at about 1 s and $0.77 per 1,000 queries, and the small local model reaches 78.5% at about 240 ms and no token cost, with more false positives on attack-like queries.

Batched auditing gave neither model a detection benefit in this setup. These rules come from a 213-item corpus built around one schema and should be checked on local traffic before deployment.

VIII. Limitations and Future Work

Several limits bound these results. First, each position bundles a prompt, an output format, and a batch size, so the study does not separate deployment position from prompt format; an ablation that crosses the three would. Second, no rule-based, classical machine learning, or transformer baseline was run on this corpus, so the study does not show that a prompted LLM outperforms established detectors. Third, the evaluation covers one generator, one frontier detector, one small detector, and one five-table schema. A different small model may not show the same position sensitivity, and the shared blind spot may reflect this generator's repertoire. Fourth, one annotator labeled the corpus and no query was executed. A second annotator with an agreement statistic and execution-based checks would test the labels, and the false positive results depend on them most. Fifth, most post-hoc batches held a single class, unlike a production log. Sixth, the corpus has 213 items, the per-family estimates are coarse, and nothing was tested at production volume or under concurrent load. Finally, the 12 payloads cannot separate an injected instruction from the attack that carries it. A matched-pairs design, the same base query with and without the instruction and weighted toward borderline bases, is the planned test of instruction-driven evasion.

IX. Conclusion

This study asked whether a prompted LLM can serve as a SQLi detector on LLM-generated queries and whether its deployment position changes the answer. Across 3,834 verdicts, the frontier model detected attacks at a steady rate at every position (80.7% to 84.4%) but flagged more attack-like benign queries in batched auditing (22.7% against 6.1% at review-time). The small local model detected well only as a runtime gatekeeper (78.5% against 51.9% elsewhere), where it came within six percentage points of the frontier model at no token cost. Both models missed many attacks in the families whose syntax resembles legitimate SQL. On this corpus the effect of deployment position depends on model capability, and a small model at the gatekeeper is a low-cost option that warrants testing at production volume and against trained baselines. To support reproducibility, the corpus, the run logs, and the code for the experiments will be made publicly available at https://github.com/msaqer/position-matters-sqli.

Acknowledgment

The author designed the study, labeled the data, ran the experiments, and checked every reported number against the run logs. Claude (Anthropic), a generative AI assistant, was used under the author's direction to help draft and edit text in all sections, to help write the experiment and analysis code, and to prepare the LaTeX source, including the tables and figures. The author reviewed all AI-assisted material and is solely responsible for the content.

References

[1] R. Pedro, M. E. Coimbra, D. Castro, P. Carreira, and N. Santos, "Prompt-to-SQL injections in LLM-integrated web applications: Risks and defenses," in *Proc. IEEE/ACM 47th Int. Conf. Softw. Eng. (ICSE)*, 2025, pp. 1768–1780.

[2] OWASP Foundation, "OWASP top 10:2025 – A05:2025 injection," https://owasp.org/Top10/2025/A05_2025-Injection/, 2025, accessed: Aug. 20, 2026.

[3] E. Kim and S. Lee, "SQL injection in LLM-generated queries: Systematic analysis of detection gaps and security risks," *IEEE Access*, vol. 14, pp. 12 797–12 815, 2026.

[4] M. Abusaqer and J. Saquer, "A comparative analysis of transformer and traditional ML models for cyberbullying detection on Twitter (now X)," in *Proc. IEEE 49th Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, 2025, pp. 1607–1612.

[5] M. Abusaqer, M. B. Senouci, and K. Magel, "Twitter user sentiments analysis: Health system cyberattacks case study," in *Proc. Int. Conf. Artif. Intell. Inf. Commun. (ICAIIC)*, 2023, pp. 18–24.

[6] M. Abusaqer, "Calibrated membership inference auditing of NLP text classifiers," in *Proc. IEEE 38th Int. Conf. Tools Artif. Intell. (ICTAI)*, Boca Raton, FL, USA, 2026, to appear.

[7] W. G. J. Halfond, J. Viegas, and A. Orso, "A classification of SQL injection attacks and countermeasures," in *Proc. IEEE Int. Symp. Secure Softw. Eng. (ISSSE)*, 2006, pp. 13–15.

[8] M. Alghawazi, D. Alghazzawi, and S. Alarifi, "Detection of SQL injection attack using machine learning techniques: A systematic literature review," *J. Cybersecur. Priv.*, vol. 2, no. 4, pp. 764–777, 2022.

[9] Z. Gui, E. Wang, B. Deng, M. Zhang, Y. Chen, S. Wei, W. Xie, and B. Wang, "SqliGPT: Evaluating and utilizing large language models for automated SQL injection black-box detection," *Appl. Sci.*, vol. 14, no. 16, p. 6929, 2024.

[10] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2022, pp. 754–768.

[11] M. Lin, H. Zhang, J. Lao, R. Li, Y. Zhou, C. Yang, Y. Cao, and M. Tang, "Are your LLM-based text-to-SQL models secure? Exploring SQL injection via backdoor attacks," *Proc. ACM Manag. Data*, vol. 3, no. 6, pp. 297:1–297:27, 2025.

[12] F. Nourmohammadzadeh Motlagh, M. Hajizadeh, M. Majd, P. Najafi, F. Cheng, and C. Meinel, "When prompts become payloads: A framework for mitigating SQL injection attacks in large language model-driven applications," arXiv:2605.10176, 2026.

[13] F. Perez and I. Ribeiro, "Ignore previous prompt: Attack techniques for language models," in *NeurIPS 2022 Workshop on Machine Learning Safety*, 2022, arXiv:2211.09527.

[14] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in *Proc. 16th ACM Workshop Artif. Intell. Secur. (AISec)*, 2023, pp. 79–90.

[15] Y. Liu, Y. Jia, R. Geng, J. Jia, and N. Z. Gong, "Formalizing and benchmarking prompt injection attacks and defenses," in *Proc. 33rd USENIX Secur. Symp.*, 2024, pp. 1831–1847.

[16] J. Shi, Z. Yuan, Y. Liu, Y. Huang, P. Zhou, L. Sun, and N. Z. Gong, "Optimization-based prompt injection attack to LLM-as-a-judge," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2024, pp. 660–674.

[17] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, "Judging LLM-as-a-judge with MT-Bench and Chatbot Arena," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), Datasets and Benchmarks Track*, vol. 36, 2023.

[18] E. B. Wilson, "Probable inference, the law of succession, and statistical inference," *J. Amer. Statist. Assoc.*, vol. 22, no. 158, pp. 209–212, 1927.

[19] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, 1947.