CoEvo-SQLi: Co-Evolving SQL Injection Detectors via Schema-Grounded Attack Synthesis

Anonymous authors
Paper under double-blind review

Abstract

Pre-execution SQL injection (SQLi) detection must block malicious queries without disrupting benign workloads and keep sensitive queries and schemas within a local trust boundary, requirements that existing defenses only partially meet. Rule- and signature-based systems are lightweight but brittle under obfuscation and semantics-preserving rewrites; learned and LLM-based detectors capture richer query patterns but are typically trained once on a fixed corpus. For small local detectors, such static training raises two problems: variants outside the fixed corpus cannot be absorbed without full retraining, and an undifferentiated budget is spent largely on already-learned, conspicuous patterns while underrepresented variants with unresolved errors receive little. We address both problems with CoEvo-SQLi, a feedback-directed co-evolutionary training framework coupling attack synthesis with multi-round detector updates. In each round, an Attacker synthesizes SQLi examples, guided by Verifier signals indicating which injection clusters should receive more training budget and where to explore further variants; the Defender fine-tunes on the resulting training slice and runs inference on a fixed validation set, whose errors the Verifier converts into next-round feedback. These coupled updates let the Attacker target the Defender's current weaknesses while the Defender learns from the resulting variants. We also introduce SQLi-bench, a schema-grounded benchmark of 48 SQLi clusters for controlled training and evaluation across diverse attack forms. Under a matched training-example budget, CoEvo-SQLi achieves a Matthews correlation coefficient (MCC, $\times 100$) of 92.5 on SQLi-bench, versus 78.0 for an identically configured static fine-tuning baseline; removing feedback-directed synthesis or payload mutation costs 10.5 and 9.2 points, respectively. Its query-only version attains the highest mean MCC on SQLi-bench, rbsqli, and WAF-A-MoLe.

1 Introduction

SQL injection (SQLi) remains a persistent threat to database-backed applications: attacker-controlled input can alter a rendered SQL query and make it deviate from the developer's intended logic before execution (Halfond et al., 2006; OWASP Foundation, n.d.; 2025b). Database access is increasingly mediated by web APIs and agentic software, which makes pre-execution screening relevant (OWASP Foundation, 2023; Li et al., 2026; Zhang et al., 2025). We study pre-execution SQLi detection at the database boundary, where a detector must identify malicious queries with high recall while keeping false alarms low enough for benign workloads. Rule- and signature-based defenses can cover known patterns efficiently, but their coverage can be reduced by obfuscation and semantics-preserving transformations (OWASP ModSecurity Project, n.d.; OWASP Foundation, 2025a; libinjection project, n.d.; Appelt et al., 2018; Demetrio et al., 2020; Floris et al., 2025). Learned detectors capture query patterns beyond explicit signatures (Uwagbole et al., 2017; Peralta-Garcia et al., 2024; Abaimov & Bianchi, 2019; Xie et al., 2019; Li et al., 2024; Lo et al., 2025), while LLM-based detectors have explored prompting and instruction tuning for SQLi detection (Zhang et al., 2024; Tian et al., 2024).

Despite these capabilities, learned detectors are often trained once on a fixed corpus, a limitation especially acute for SQLi, where static training raises two problems. First, unlike standard learned

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

tasks such as translation or image classification, SQLi is shaped by an active adversary: attack studies show that payload syntax and structure can be mutated to evade deployed defenses (Appelt et al., 2018; Demetrio et al., 2020; Floris et al., 2025). Second, training value is uneven: conspicuous patterns can be learned quickly, whereas failures on concealed variants reveal unresolved weaknesses. Training indiscriminately on the full pool can therefore spend a limited budget on already learned patterns while neglecting those weaknesses. These two problems motivate a training framework that, within a fixed budget, directs training toward the detector's current weaknesses and explores variants beyond the initial library.

To address both problems, we propose CoEvo-SQLi (Figure 1), a feedback-directed framework that couples attack synthesis with multi-round training of a small LLM-based detector. Its central idea is to use the detector's current failures to decide which injection clusters to explore and how to allocate the next round's training budget, while suitably prompted mutation recombines attack knowledge into new variants within the selected clusters (Appelt et al., 2014; Liu et al., 2020; Gui et al., 2024). An Attacker synthesizes SQLi examples and applies payload mutation to explore beyond the initial library; a Defender fine-tunes on the resulting training slice; a Verifier evaluates the updated detector on a fixed validation set. From the validation errors it derives two control signals: cluster weights, which steer subsequent attack selection and mutation while preserving a nonzero sampling probability for every cluster, and a benign-example sampling ratio for each training slice. These coupled updates yield co-evolution: the Attacker adapts its synthesis policy to the Defender's weaknesses, and the Defender learns from the resulting examples.

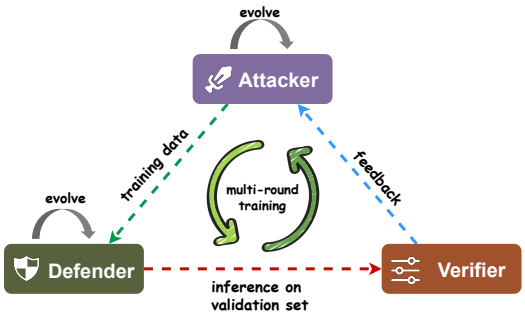

Figure 1: The CoEvo-SQLi loop: the Attacker, Defender, and Verifier co-evolve across training rounds.

To support this process, we introduce SQLi-bench, which organizes attack examples into 48 injection clusters with controlled training, validation, and held-out evaluation splits; its paired schema-grounded and query-only settings support evaluating adaptive training across diverse attack forms and quantifying the contribution of schema context, and its construction pipeline also enables dynamic synthesis during CoEvo-SQLi training.

Under a fixed training-example budget, CoEvo-SQLi achieves an MCC of 92.5 ($\times 100$) on schema-grounded SQLi-bench, outperforming Static SFT (78.0) and all closed-source baselines; removing feedback or payload mutation costs 10.5 and 9.2 points, respectively, indicating complementary contributions. The query-only version also attains the highest mean MCC on SQLi-bench, rbsqli, and WAF-A-MoLe, suggesting that the framework generalizes beyond schema-grounded SQLi-bench.

Our contributions are threefold: (i) CoEvo-SQLi, a training framework coupling feedback-directed attack synthesis with iterative detector updates, where adaptive cluster selection directs training budget toward unresolved attack forms and payload mutation explores variants beyond the initial library; (ii) SQLi-bench, a schema-grounded benchmark with 48 injection clusters and controlled data splits, together with a construction pipeline shared by benchmark generation and adaptive training; and (iii) a controlled evaluation under matched training-example budgets that isolates the contributions of feedback, payload mutation, and schema context, and characterizes the associated training and inference costs.

2 Task Formulation and Taxonomy

We first formulate schema-grounded SQLi detection and then define the taxonomy that connects data construction, adaptive synthesis, and evaluation.

2.1 SCHEMA-GROUNDED SQLI DETECTION

The target schema, obtained from database metadata or application-maintained definitions, provides structural context about tables, columns, data types, and relationships that may help distinguish malicious queries from benign ones. We therefore consider schema-grounded detection, in which the detector receives both the rendered SQL query and its target schema; Appendix A gives an illustrative example and further discussion. We formulate the task as binary classification: let $\mathcal{Q}$ denote the space of rendered SQL queries, $\mathcal{S}$ the space of target schemas represented as DDL, and $\mathcal{Y} = \{\texttt{malicious}, \texttt{benign}\}$ the label space; each example is a triple $(q, s, y) \in \mathcal{Q} \times \mathcal{S} \times \mathcal{Y}$, and a schema-grounded detector $f_\theta : \mathcal{Q} \times \mathcal{S} \to \mathcal{Y}$ predicts $\hat{y} = f_\theta(q, s)$.

Query-only detection considers the same classification task but omits the target schema: given only $q \in \mathcal{Q}$, a query-only detector $g_\phi : \mathcal{Q} \to \mathcal{Y}$ predicts $\hat{y} = g_\phi(q)$.

2.2 SQLI-INSTANCE TAXONOMY AND INJECTION CLUSTERS

SQLi instances differ substantially in attack category and form (Halfond et al., 2006; Sadeghian et al., 2013); treating them as an undifferentiated pool can waste training budget on redundant or already-covered examples and leaves data selection no principled basis for prioritizing difficult or underrepresented forms (Killamsetty et al., 2021). We therefore introduce a SQLi-instance taxonomy that organizes this heterogeneity into discrete units for coverage analysis and feedback-directed data selection.

We characterize each malicious example along three dimensions (injection technique τ, payload reference scope σ, and comment state κ) and assign it the cluster key $c_k = (\tau, \sigma, \kappa)$; malicious examples sharing a key form an injection cluster.

Injection technique. τ takes one of six operational families selected for our benchmark: tautology, UNION query-based, piggy-backed query, error-based, boolean-based blind, and time-based blind (Halfond et al., 2006; Clarke-Salt, 2012; Sadeghian et al., 2013; OWASP Foundation, n.d.).

Payload reference scope. σ distinguishes literal-only references (LOR), target-schema references to tables or columns (TSR), and system-catalog references to objects or variables (SCR). This dimension captures the context required to interpret a payload and separates attacks by degree of schema dependence.

Comment state. κ takes one of three states: `no_comment` (no line-comment delimiter), `clean_comment` (a delimiter without trailing natural-language text), and `cepp` (natural-language text appended after the delimiter). The motivation for this dimension and its relation to comment-embedded prompt injection (CEPP) are discussed in Appendix B.

Under our construction contract, boolean-blind/LOR and piggy-backed/LOR do not form meaningful combinations and are excluded, leaving 16 technique–scope pairs; combined with three comment states, these yield 48 injection clusters, plus one cluster for benign examples. In SQLI-BENCH, the clusters define coverage units for data construction and evaluation; in COEVO-SQLI, they set the granularity at which validation errors are summarized and synthesis budgets are allocated. Appendix B details each dimension and lists representative attack patterns.

3 SQLI-BENCH: SCHEMA-GROUNDED DATA CONSTRUCTION

3.1 BENCHMARK MOTIVATION

Schema-grounded SQLi detection requires examples pairing each rendered query with the schema against which it is interpreted, but the public datasets used in our comparison provide query strings without target schemas and cannot evaluate this task directly (sajid576, n.d.; Mullick et al., 2025; Demetrio et al., 2020); many also suffer incomplete SQL, duplicate samples, and unreliable labels. We construct SQLI-BENCH to fill this input-contract and data-quality gap; Appendix C compares its data quality with the public datasets.

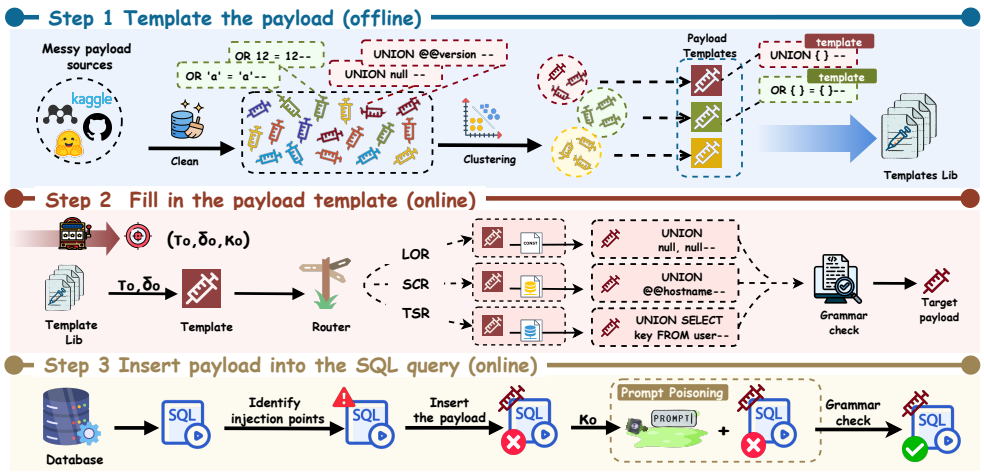

Figure 2: Construction pipeline for schema-grounded SQLi examples in SQLI-BENCH.

3.2 CONSTRUCTION PIPELINE

Step 1: Template the payload (offline). SQLi payloads vary widely in surface form but often express recurring strategies (Halfond et al., 2006; Clarke-Salt, 2012); we therefore manually clean, consolidate, and abstract payloads from sqlmap and other defensive resources into reusable templates (sqlmapproject, n.d.), each carrying an injection technique and a payload reference scope. The library is partitioned into disjoint train/validation/test pools: training observes only 63 templates (15% of the library), while held-out evaluation instantiates 180 test-side designs never seen in training.

Step 2: Fill in the payload template (online). Given a payload template and a target schema, we instantiate its placeholders according to the reference scope: literal-only templates are filled with constants, target-schema templates are bound to tables and columns of the target schema, and system-catalog templates are bound to supported metadata objects or expressions. The sampled injection cluster also fixes the comment state: none, a clean SQL comment, or a comment containing prompt-injection text.

Step 3: Insert the payload into the SQL query (online). The instantiated payload is inserted into an explicit injection point in each carrier query, and the resulting query is checked for syntactic validity. Accepted malicious samples are stored as query–schema–label triples; benign queries are not injected and keep their original schemas. The same pipeline (Figure 2) constructs the fixed SQLI-BENCH splits and dynamically synthesizes samples from a selected cluster during COEVO-SQLI training.

SQLI-BENCH comprises three fixed splits with distinct roles: the training split trains only the budget-matched Static SFT baseline; the validation split is used only within COEVO-SQLI, where the VERIFIER converts cluster-level errors into feedback for subsequent rounds; and the test split, isolated from both, is used only for final reporting of COEVO-SQLI and other baselines. Split sizes, source-pool statistics, and cluster distributions appear in Appendix B.

4 COEVO-SQLI

COEVO-SQLI is a feedback-directed, multi-round training paradigm for small LLM-based SQLi detection. It replaces one-time training on a fixed, undifferentiated corpus with two complementary mechanisms: validation feedback directs the limited synthesis budget toward unresolved attack forms with greater training value, while LLM-based payload mutation leverages the LLM's ability, under appropriate prompting, to creatively recombine attack knowledge into valid category-consistent variants beyond the initial payload library, improving detection on unseen attack forms. A sampling floor preserves coverage of other attacks and benign queries. Figure 3 shows the overall

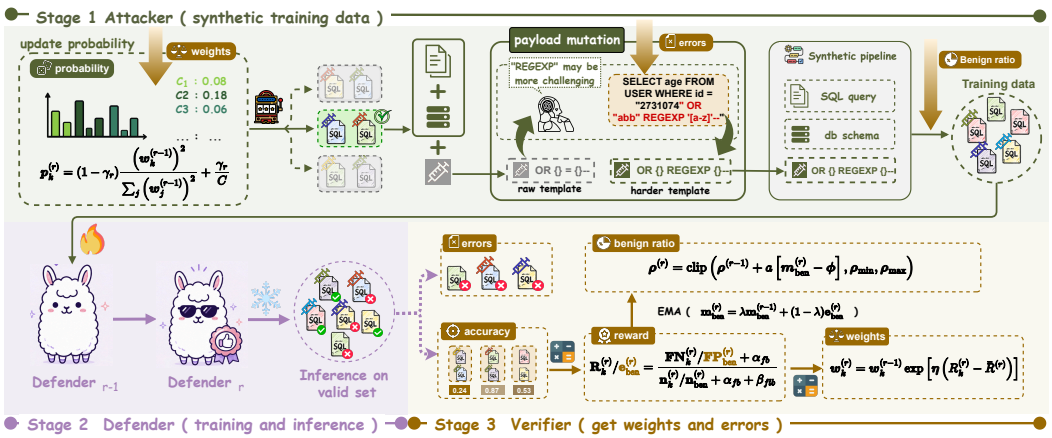

Figure 3: Overview of the ATTACKER–DEFENDER–VERIFIER loop in COEVO-SQLI.

ATTACKER–DEFENDER–VERIFIER loop: at each round, the ATTACKER selects injection clusters and synthesizes a training slice; the DEFENDER fine-tunes on it; and the VERIFIER evaluates the updated model on a fixed validation set, converting cluster-level false negatives and benign false positives into the next round's feedback.

4.1 ATTACKER: FAILURE-DIRECTED SYNTHESIS

The ATTACKER constructs a fixed-budget training slice at each round by selecting injection clusters and synthesizing examples from them. The benign sampling ratio determines the number of benign examples, with the remaining budget assigned to injection examples. Synthesis uses the train-side source pool and the construction pipeline described in Section 3.2.

Cluster selection. The ATTACKER uses cluster weights supplied by the VERIFIER to compute a sampling distribution over injection clusters. Let $w_k^{(r-1)}$ denote the weight of cluster k from the previous round, and let C denote the number of injection clusters. Using squared weights, the sampling probability at round r is

$$p_k^{(r)} = (1-\gamma_r)\frac{(w_k^{(r-1)})^2}{\sum_j (w_j^{(r-1)})^2} + \frac{\gamma_r}{C}. \tag{1}$$

Here, γ_r controls the mixture between weight-based selection and uniform exploration. The first term favors clusters with larger accumulated validation errors, while the second preserves a nonzero sampling probability for every cluster. The exploration coefficient decays linearly across rounds, gradually shifting emphasis toward the clusters identified by feedback. The ATTACKER samples K distinct clusters without replacement from this distribution to guide the current round's synthesis.

Payload mutation and synthesis. The ATTACKER applies LLM-based mutation to payload cores from the selected clusters, constrained to preserve the injection technique and reference scope. We use anti-imitation few-shot prompting: the source payload cores and previously accepted mutations from the same technique–scope category serve as examples to avoid, steering the LLM toward alternative expressions within the category constraints.

The mutation probability follows the complementary linear schedule (endpoints with Algorithm D.1): it is kept low in early rounds, so that the DEFENDER first learns the static library's coverage, and rises in later rounds to introduce more variants once the initial patterns are learned. The synthesized examples are combined with benign query–schema pairs to form the training slice.

4.2 DEFENDER: SMALL LOCAL FINE-TUNING

In the schema-grounded setting, the DEFENDER is a small language model (0.5B parameters in our experiments) that predicts whether a query is malicious or benign from the SQL text and the

target database schema rendered as DDL. At each round, it is updated using the newly generated training slice, starting from the model obtained in the previous round. Each update is standard supervised fine-tuning, minimizing the next-token loss on the label tokens, $\mathcal{L}(\theta) = -\sum_t \log p_\theta(y_t \mid x, y_{<t})$. The updated model is evaluated on the fixed validation split, and the resulting errors provide feedback for subsequent attack synthesis and benign-example allocation.

4.3 VERIFIER: ERROR-TO-FEEDBACK CONVERSION

The VERIFIER converts validation errors into two feedback signals passed to the ATTACKER: injection-cluster weights, which guide attack synthesis, and the next-round benign sampling ratio, which controls the benign share of training examples.

Error measurement. After each training round, the VERIFIER evaluates the updated DEFENDER on the validation set, recording false negatives per injection cluster and false positives among benign examples. Let $n_k^{(r)}$ and $n_{\text{ben}}^{(r)}$ denote the numbers of malicious examples in cluster k and of benign examples evaluated at round r. Both error rates are Laplace-smoothed to avoid extreme estimates when a round contains few examples:

$$R_k^{(r)} = \frac{\text{FN}_k^{(r)} + 1}{n_k^{(r)} + 2}, \qquad e_{\text{ben}}^{(r)} = \frac{\text{FP}_{\text{ben}}^{(r)} + 1}{n_{\text{ben}}^{(r)} + 2}, \tag{2}$$

where $R_k^{(r)}$ is the smoothed false-negative rate for cluster k and $e_{\text{ben}}^{(r)}$ the smoothed benign false-positive rate.

Cluster-weight update. The VERIFIER applies a centered multiplicative-weights update, a Hedge-style construction (Freund & Schapire, 1997), that raises the relative weight of clusters whose smoothed false-negative rates exceed the current average. Let $w_k^{(r-1)}$ denote the previous weight of cluster k and $\bar{R}^{(r)}$ the mean smoothed false-negative rate across clusters; the weights are updated as

$$w_k^{(r)} = w_k^{(r-1)} \exp\left[\eta \left(R_k^{(r)} - \bar{R}^{(r)}\right)\right], \tag{3}$$

where η controls the update strength.

Benign-ratio update. To control false positives, the VERIFIER maintains an exponential moving average (EMA) $m_{\text{ben}}^{(r)}$ of the smoothed benign false-positive rate and adjusts the benign sampling ratio $\rho^{(r)}$:

$$m_{\text{ben}}^{(r)} = \lambda m_{\text{ben}}^{(r-1)} + (1-\lambda) e_{\text{ben}}^{(r)}, \tag{4}$$

$$\rho^{(r)} = \text{clip}\Big(\rho^{(r-1)} + a\left[m_{\text{ben}}^{(r)} - \phi\right], \rho_{\min}, \rho_{\max}\Big). \tag{5}$$

Here, λ controls how strongly the EMA retains previous error history, ϕ is the target smoothed benign error, and a the adjustment size. When the EMA exceeds the target, the benign share of the next slice increases; when it falls below, the share decreases and more budget goes to injection examples.

5 EXPERIMENTS

We organize the evaluation around four questions: (Q1) Does COEVO-SQLI improve schema-grounded SQLi detection under a matched training-example budget? (Q2) How do feedback and payload mutation contribute? (Q3) How much does schema context contribute, and does COEVO-SQLI generalize beyond SQLI-BENCH on public query-only benchmarks? (Q4) What training, synthesis, and inference costs does it incur? Detailed protocols and complete results appear in Appendices E and F.

5.1 EXPERIMENTAL SETUP

Data and training. We use the fixed SQLI-BENCH splits of Section 3.2. The DEFENDER is Qwen2.5-Coder-0.5B-Instruct (Hui et al., 2024), adapted with LoRA (Hu et al., 2022). COEVO-

SQLI and Static SFT share the same fine-tuning configuration and matched training-example budgets; benchmark composition and implementation settings are in the appendix.

Comparators. In the schema-grounded setting, we compare COEVO-SQLI with Static SFT and seven closed-source LLMs (claude-sonnet-5, gpt-5.6-luna, kimi-k2.7-code, glm-5.3-flash, deepseek-v4-flash, qwen3.8-flash, and gemini-3.8-flash) under fixed zero-shot prompts; exact identifiers, prompts, and decoding settings are in the appendix. The untrained backbone labels almost every query malicious and provides almost no discrimination, so we omit it from the main text (Appendix F.1).

The three public resources used in our complementary evaluation are query-only: SQLiV3 (sajid576, n.d.), rbsqli (Mullick et al., 2025), and WAF-A-MoLe (Demetrio et al., 2020). They test generalization beyond SQLI-BENCH and enable comparisons with existing machine-learning and rule-based detectors under their native input contract; paired schema-grounded and query-only runs on SQLI-BENCH quantify the contribution of schema context. API identifiers and implementation details for these baselines are in Appendix E.

Metrics. We report the Matthews correlation coefficient (MCC) as the primary aggregate metric, together with attack recall (true-positive rate, TPR) and the benign false-positive rate (FPR; lower is better); figures and tables report MCC $\times 100$. Locally trained conditions use three seeds; SDs and seed-paired tests appear in Appendices F and F.2, and API, rule-based, and ML baselines use one run each.

5.2 SCHEMA-GROUNDED SQLI DETECTION (Q1)

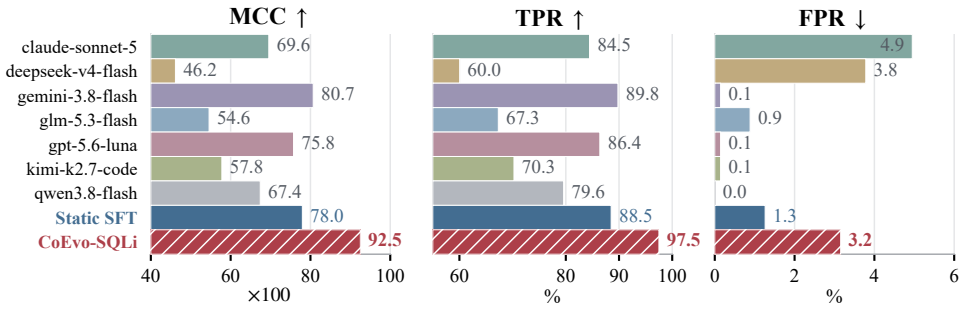

Figure 4: Schema-grounded detection on SQLI-BENCH. Hatching highlights COEVO-SQLI.

Under the matched budget, COEVO-SQLI substantially improves schema-grounded detection on SQLI-BENCH (Figure 4), reaching an MCC of 92.5 and outperforming Static SFT and the strongest closed-source baseline, gemini-3.8-flash. The improvement over Static SFT is driven by attack recall: TPR increases from 88.5% to 97.5%, while FPR increases from 1.3% to 3.2%, recovering far more missed attacks than the benign false alarms added.

The gains concentrated where Static SFT left larger gaps (Table 8 in Appendix F.1): cluster-macro TPR improves by 7.2 points for error-based attacks, 6.2 for inputs without comments, and 4.1 for literal-only references, while near-saturated categories such as clean-comment inputs have little room to improve, consistent with feedback steering training toward the clusters that remain difficult.

5.3 FEEDBACK AND MUTATION ABLATION (Q2)

Under the same budget, both components contribute (Figure 5): removing feedback or mutation reduces MCC by 10.5 and 9.2 points, respectively. The combination also improves the recall–false-alarm balance over either component alone: versus feedback-only, TPR rises from 94.8% to 97.5% while FPR falls from 9.2% to 3.2%; versus mutation-only, it is better on both metrics, with the highest MCC of 92.5.

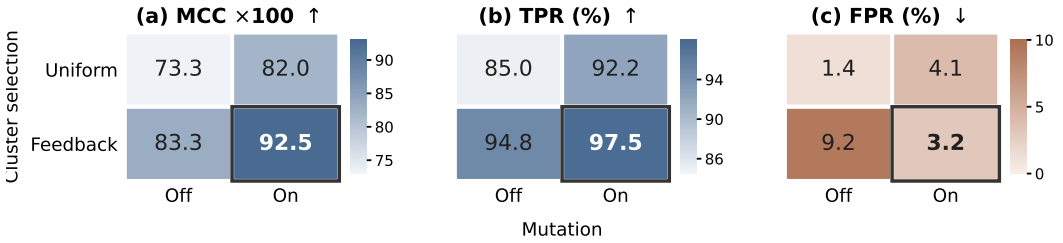

Figure 5: The 2×2 feedback–mutation ablation. Panels report MCC $\times 100$, TPR (%), and FPR (%), respectively.

5.4 Schema Contribution and Generalization Beyond SQLi-bench (Q3)

CoEvo-SQLi (QO) (the query-only variant of CoEvo-SQLi), trained with the same framework, attains the best mean MCC on SQLi-bench and on rbsqli and WAF-A-MoLe, two of the three public benchmarks (Table 1), while maintaining useful attack recall at low false-alarm rates. Its weaker SQLiV3 result relative to the strongest API baselines likely reflects a format mismatch: our formulation assumes complete SQL queries, whereas SQLiV3 contains many incomplete inputs (Appendix C).

The comparison also exposes the limits of traditional detectors. ModSecurity CRS performs well on some public benchmarks yet achieves only 36.0 MCC on SQLi-bench; the ML baselines trained on SQLi-bench, Trident and EP-CNN, show limited in-distribution performance and do not transfer consistently to the public benchmarks. Both patterns suggest that fixed rules and conventional feature-based models struggle with the semantic, task-dependent patterns that vary across datasets.

Table 1: Query-only detection and zero-adaptation transfer. Each cell reports MCC $\times 100$, TPR (%), and FPR (%). Learned detectors are trained on SQLi-bench and evaluated on each benchmark without further adaptation. Bold and underlined values indicate the best and second-best results in each column.

Model	SQLi-bench			rbsqli			SQLiV3			WAF-A-MoLe		
	MCC	TPR	FPR	MCC	TPR	FPR	MCC	TPR	FPR	MCC	TPR	FPR
ModSecurity CRS	36.0	53.6	9.9	63.8	47.4	**0.0**	82.3	81.9	2.5	84.7	86.2	1.5
Trident	42.4	82.8	36.8	0.1	59.9	59.7	2.3	33.7	31.5	26.1	73.8	48.3
EP-CNN	26.4	89.8	66.8	15.7	3.2	**0.0**	-2.6	28.5	30.9	67.1	74.2	7.6
claude-sonnet-5	61.7	74.3	0.1	5.9	99.1	96.8	**93.5**	95.5	2.2	73.9	**93.8**	21.1
deepseek-v4-flash	46.6	60.0	3.2	44.8	87.9	35.5	83.7	96.6	10.8	56.4	86.8	31.8
gemini-3.8-flash	71.7	83.2	**0.0**	58.2	**100.0**	31.9	90.2	**99.2**	7.1	82.8	91.8	9.0
glm-5.3-flash	57.6	70.7	0.9	11.9	99.8	93.9	84.8	**99.2**	11.9	55.7	91.6	39.2
gpt-5.6-luna	67.7	80.2	0.6	54.4	**100.0**	36.3	88.5	98.3	7.9	76.6	90.2	13.7
kimi-k2.7-code	56.3	69.5	1.2	44.8	99.5	48.2	88.3	98.3	8.1	65.3	91.1	27.4
qwen3.8-flash	66.2	78.6	0.1	17.9	99.9	87.6	87.8	98.3	8.6	72.7	92.5	20.8
Static SFT	77.1	87.8	1.0	92.5	88.6	**0.0**	65.8	55.5	**0.4**	84.4	85.9	1.5
CoEvo-SQLi (QO)	**81.5**	**91.0**	2.0	**93.1**	93.4	1.3	77.1	75.6	3.0	**87.9**	89.1	**1.4**

5.5 Inference Efficiency and Training Cost (Q4)

CoEvo-SQLi serves inference exactly like Static SFT, with the same backbone, prompt structure, and serving configuration, so latency is essentially identical under the terminal batch-size-one evaluation: about 29 ms per query (34 queries/s) in the schema-grounded setting and 17–18 ms (57–59 queries/s) in the query-only setting, the latter faster because schema descriptions are omitted; detailed measurements are in Appendix F.6.

The extra cost is offline, in iterative synthesis and training: the complete eight-round schema-grounded pipeline takes 35.4 minutes versus 5.9 for Static SFT at the same 2,400-example budget, with attack synthesis and fine-tuning dominating the added time and mutation accounting for most of the synthesis cost; query-only training reduces the total to 30.1 minutes. Overall, CoEvo-SQLi adds a moderate offline cost while preserving inference efficiency.

6 Related Work

SQLi detection. Existing SQLi detectors broadly fall into rule-based and learned families (Halfond et al., 2006). Signature-based systems such as ModSecurity with the OWASP Core Rule Set and token-based analyzers such as libinjection provide rule-driven coverage, but evolutionary and adversarial evaluations show that obfuscation and semantics-preserving transformations can expose coverage gaps (OWASP ModSecurity Project, n.d.; OWASP Foundation, 2025a; libinjection project, n.d.; Appelt et al., 2018; Demetrio et al., 2020; Floris et al., 2025). Learned detectors capture query patterns beyond explicit signatures, using classical classifiers (Uwagbole et al., 2017; Peralta-Garcia et al., 2024), convolutional models such as CODDLE and EP-CNN (Abaimov & Bianchi, 2019; Xie et al., 2019), and structural or contextual representations such as Trident and lightweight self-attention (Li et al., 2024; Lo et al., 2025). More recently, LLM-based approaches have explored prompting and instruction tuning for SQLi detection (Zhang et al., 2024; Tian et al., 2024). These fixed-corpus evaluations leave robustness to subsequently emerging variants difficult to ensure (Demetrio et al., 2020; Floris et al., 2025).

SQLi data synthesis and vulnerability exploration. Automated SQLi testing generates attack inputs to expose vulnerabilities and defense weaknesses. Ardilla and μ4SQLi construct concrete exploits through taint tracking and mutation operators (Kiezun et al., 2009; Appelt et al., 2014); DeepSQLi, SQIRL, and SqliGPT explore neural, reinforcement-learning, and LLM-based generation (Liu et al., 2020; Al Wahaibi et al., 2023; Gui et al., 2024); ML-Driven and WAF-A-MoLe show that feedback-guided search and mutation uncover evasive payloads (Appelt et al., 2018; Demetrio et al., 2020); ModSec-AdvLearn further incorporates adversarial examples into training to strengthen WAF robustness (Floris et al., 2025).

Adaptive training-data selection. Adaptive data selection allocates a limited training budget to examples most useful for the current model. Guided SQLi evasion studies show targeted mutation can uncover weaknesses more efficiently than unguided search (Demetrio et al., 2020), motivating (rather than evidencing) heterogeneous training value across SQLi examples. For model training, GRAD-MATCH selects subsets through gradient matching (Killamsetty et al., 2021), and instruction-tuning methods select informative tasks and examples via prompt uncertainty, learning progress, or dynamic utility estimates (Kung et al., 2023; Mekala et al., 2024; Lin et al., 2025); these works establish the broader data-selection setting that our feedback-directed SQLi procedure instantiates.

7 Conclusion

We present CoEvo-SQLi, a feedback-directed training framework for small local SQLi detectors, and SQLi-bench, a schema-grounded benchmark covering 48 injection clusters. Under a matched training-example budget, CoEvo-SQLi improves MCC over Static SFT by 14.5 points on schema-grounded SQLi-bench, with complementary contributions from feedback and payload mutation, and its query-only version attains the highest mean MCC on SQLi-bench, rbsqli, and WAF-A-MoLe. Schema context improves detection for most evaluated models. Future work will explore broader SQL dialects and real-world workloads.

AI Use Statement

This work uses generative AI in two ways. As part of the research method, we used an auxiliary open-weight LLM (Qwen3.6-27B; Appendix E.3) for taxonomy-conditioned payload mutation during training-data synthesis, and we evaluated seven API-accessed LLMs as zero-shot baselines under the protocols described in Section 5. For manuscript preparation, generative AI tools were used to assist with rewriting, organization, and proofreading. The authors reviewed all AI-assisted

text, code, data-generation outputs, and experimental claims, and take responsibility for the final content of the paper.

Ethics Statement

This work studies SQL injection detection and therefore has dual-use implications. The generated payloads, taxonomy, and benchmark are intended to improve defensive screening and robustness evaluation, but the same artifacts could inform attack construction if released without safeguards. Any public release should therefore avoid exposing operationally harmful payload details beyond what is already present in standard educational or defensive resources, document intended defensive use, and separate reproducibility materials from deployment-sensitive configurations.

Reproducibility Statement

An anonymized repository containing the generation scripts, split metadata, prompts, training configuration, inference scripts, benchmark manifest, run manifests, and evaluation code is provided in the supplementary material and at `https://anonymous.4open.science/r/CoEvo-SQLi-187/`, subject to the security release safeguards described in the Ethics Statement.

References

Stanislav Abaimov and Giuseppe Bianchi. CODDLE: Code-injection detection with deep learning. *IEEE Access*, 7:128617–128627, 2019. doi: 10.1109/ACCESS.2019.2939870.

Salim Al Wahaibi, Myles Foley, and Sergio Maffeis. SQIRL: Grey-box detection of SQL injection vulnerabilities using reinforcement learning. In *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 6097–6114, Anaheim, CA, USA, 2023. USENIX Association. URL `https://www.usenix.org/conference/usenixsecurity23/presentation/al-wahaibi`.

Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. Automated testing for SQL injection vulnerabilities: An input mutation approach. In *Proceedings of the 23rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 259–269. ACM, 2014. doi: 10.1145/2610384.2610403.

Dennis Appelt, Cu Duy Nguyen, Annibale Panichella, and Lionel C. Briand. A machine-learning-driven evolutionary approach for testing web application firewalls. *IEEE Transactions on Reliability*, 67(3):733–757, 2018. doi: 10.1109/TR.2018.2805763.

Justin Clarke-Salt. *SQL Injection Attacks and Defense*. Syngress, Waltham, MA, USA, 2nd edition, 2012. ISBN 978-1-59749-963-7. URL `https://shop.elsevier.com/books/sql-injection-attacks-and-defense/clarke-salt/978-1-59749-963-7`.

Luca Demetrio, Andrea Valenza, Gabriele Costa, and Giovanni Lagorio. WAF-A-MoLe: Evading web application firewalls through adversarial machine learning. In *Proc. ACM Symposium on Applied Computing (SAC)*, pp. 1745–1752, New York, NY, USA, 2020. ACM. doi: 10.1145/3341105.3373962.

Giuseppe Floris, Christian Scano, Biagio Montaruli, Luca Demetrio, Andrea Valenza, Luca Compagna, Davide Ariu, Luca Piras, Davide Balzarotti, and Battista Biggio. ModSec-AdvLearn: Countering adversarial SQL injections with robust machine learning. *IEEE Transactions on Information Forensics and Security*, 20:6693–6705, 2025. doi: 10.1109/TIFS.2025.3583234. URL `https://arxiv.org/abs/2308.04964`.

Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997. doi: 10.1006/jcss.1997.1504.

Zhiwen Gui, Enze Wang, Binbin Deng, Mingyuan Zhang, Yitao Chen, Shengfei Wei, Wei Xie, and Baosheng Wang. SqliGPT: Evaluating and utilizing large language models for automated SQL injection black-box detection. *Applied Sciences*, 14(16):6929, 2024. doi: 10.3390/app14166929.

William G. J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of SQL injection attacks and countermeasures. In *2006 IEEE International Symposium on Secure Software Engineering (ISSSE)*, Arlington, VA, USA, 2006. IEEE. URL https://dblp.org/rec/conf/isse3/HalfondVO06.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, Virtual, 2022. OpenReview.net. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2.5-Coder technical report. *CoRR*, abs/2409.12186, 2024. doi: 10.48550/arXiv.2409.12186.

Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pp. 199–209, Vancouver, Canada, 2009. IEEE. URL https://homes.cs.washington.edu/~mernst/pubs/create-attacks-icse2009-abstract.html.

Krishnateja Killamsetty, Durga Sivasubramanian, Ganesh Ramakrishnan, Abir De, and Rishabh Iyer. GRAD-MATCH: Gradient matching based data subset selection for efficient deep model training. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139, pp. 5464–5474. PMLR, 2021. URL https://proceedings.mlr.press/v139/killamsetty21a.html.

Po-Nien Kung, Fan Yin, Di Wu, Kai-Wei Chang, and Nanyun Peng. Active instruction tuning: Improving cross-task generalization by training on prompt sensitive tasks. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 1813–1829, Singapore, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.112. URL https://aclanthology.org/2023.emnlp-main.112/.

Boyan Li, Yiran Peng, Yupeng Xie, Sirong Lu, Yizhang Zhu, Xing Mu, Xinyu Liu, and Yuyu Luo. DeepEye: A steerable self-driving data agent system. In *Companion of the 2026 International Conference on Management of Data*, SIGMOD Companion '26, pp. 74–77, Bengaluru, India, 2026. ACM. doi: 10.1145/3788853.3801612.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM already serve as a database interface? a BIg bench for large-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 42330–42357, New Orleans, LA, USA, 2023. Neural Information Processing Systems Foundation, Inc. doi: 10.52202/075280-1835.

Yuanlin Li, Zhiwei Xu, Min Zhou, Hai Wan, and Xibin Zhao. Trident: Detecting SQL injection attacks via abstract syntax tree-based neural network. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 2225–2229, Sacramento, CA, USA, 2024. ACM. doi: 10.1145/3691620.3695289.

libinjection project. libinjection: SQL / SQLi tokenizer parser analyzer. https://github.com/libinjection/libinjection, n.d. Accessed: 2026-09-26.

Xiaotian Lin, Yanlin Qi, Yizhang Zhu, Themis Palpanas, Chengliang Chai, Nan Tang, and Yuyu Luo. LEAD: Iterative data selection for efficient LLM instruction tuning. *Proceedings of the VLDB Endowment*, 19(3):426–439, 2025. doi: 10.14778/3778092.3778103.

Muyang Liu, Ke Li, and Tao Chen. DeepSQLi: Deep semantic learning for testing SQL injection. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 286–297, New York, NY, USA, 2020. ACM. doi: 10.1145/3395363.3397375.

Rui-Teng Lo, Wen-Jyi Hwang, and Tsung-Ming Tai. SQL injection detection based on lightweight multi-head self-attention. *Applied Sciences*, 15(2):571, 2025. doi: 10.3390/app15020571.

Dheeraj Mekala, Alex Nguyen, and Jingbo Shang. Smaller language models are capable of selecting instruction-tuning training data for larger language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 10456–10470, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.623. URL https://aclanthology.org/2024.findings-acl.623/.

Mohammad Abu Obaida Mullick, Rezaur Rahman Ratul, Sama Binte Sharif, Sanjida Jannat Anannaya, and Mir Moynuddin Ahmed Shibly. Rule-based SQL injection (RbSQLi) dataset. Mendeley Data, Version 2, 2025. URL https://data.mendeley.com/datasets/xz4d5zj5yw/2. Published: 2025-05-27; accessed: 2026-09-26.

OWASP Foundation. OWASP api security top 10 – 2023. https://owasp.org/projects/api-security-project, 2023. Accessed: 2026-09-26.

OWASP Foundation. OWASP ModSecurity Core Rule Set. https://owasp.org/www-project-modsecurity-core-rule-set/, 2025a. Accessed: 2026-09-26.

OWASP Foundation. OWASP top 10:2025 – a05:2025 injection. https://top10.owasp.org/2025/A05_2025-Injection/, 2025b. Accessed: 2026-09-26.

OWASP Foundation. Testing for SQL injection. https://wstg.owasp.org/v4.2/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection/, n.d. Accessed: 2026-09-26.

OWASP ModSecurity Project. ModSecurity: Open Source Web Application Firewall. https://github.com/owasp-modsecurity/ModSecurity, n.d. Accessed: 2026-09-26.

Edwin Peralta-Garcia, Juan Quevedo-Monsalbe, Victor Tuesta-Monteza, and Juan Arcila-Diaz. Detecting structured query language injections in web microservices using machine learning. *Informatics*, 11(2):15, 2024. doi: 10.3390/informatics11020015.

Qwen Team. Qwen3.6-27b. Hugging Face, 2026. URL https://huggingface.co/Qwen/Qwen3.6-27B. Open-weight model release; accessed: 2026-09-26.

Amirmohammad Sadeghian, Mazdak Zamani, and Shahidan M. Abdullah. A taxonomy of SQL injection attacks. In *Proceedings of the International Conference on Informatics and Creative Multimedia (ICICM)*, pp. 269–273, Kuala Lumpur, Malaysia, 2013. IEEE. doi: 10.1109/ICICM.2013.53.

sajid576. Sql injection dataset (SQLiV3). Kaggle, n.d. URL https://www.kaggle.com/datasets/sajid576/sql-injection-dataset. Version and original upload date not stated by uploader; accessed: 2026-09-26.

sqlmapproject. sqlmap: Automatic SQL injection and database takeover tool. https://github.com/sqlmapproject/sqlmap, n.d. Accessed: 2026-09-26.

Zonghang Tian, Shuting Lou, Yingying Zhang, Zhengdan Jiang, and Xingkai Cheng. SQL injection vulnerability detection based on pissa-tuned llama 3 large language model. In *Proc. IEEE International Conference on Frontier Technologies of Information and Computer (ICFTIC)*, pp. 255–259, Qingdao, China, 2024. IEEE. doi: 10.1109/ICFTIC64248.2024.10912886.

Solomon O. Uwagbole, William J. Buchanan, and Lu Fan. Applied machine learning predictive analytics to SQL injection attack detection and prevention. In *Proc. IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 1087–1090, Lisbon, Portugal, 2017. IEEE. doi: 10.23919/INM.2017.7987433.

Xin Xie, Chunhui Ren, Yusheng Fu, Jie Xu, and Jinhong Guo. SQL injection detection for web applications based on elastic-pooling CNN. *IEEE Access*, 7:151475–151481, 2019. doi: 10.1109/ACCESS.2019.2947527.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *ICLR*. OpenReview.net, 2025.

Yingying Zhang, Zhengdan Jiang, Xingkai Cheng, Han Wu, Zonghang Tian, and Shuting Lou. A method of SQL injection attack detection based on large language models. In *Proc. IEEE International Conference on Computer Network Technology and Electronic and Information Engineering (CNTEIE)*, pp. 154–158, Changchun, China, 2024. IEEE. doi: 10.1109/CNTEIE66268.2024.00035.

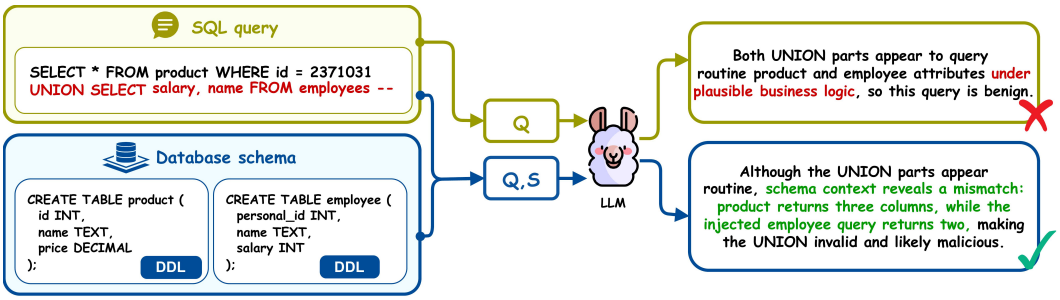

Figure 6: Schema context provides additional structural evidence for pre-execution SQLi detection. Without the target schema, the detector cannot resolve the number of columns returned by `SELECT * FROM product`. The schema reveals three columns, compared with two in `SELECT name, salary FROM employees`, exposing incompatible UNION projections.

A Why Schema Context Matters for Pre-Execution SQLi Detection

At the database boundary, the target database schema is information that a detector can access alongside the rendered SQL query. It can be obtained from database metadata or application-maintained schema definitions and supplied as additional input. However, existing query-only SQLi detection settings largely leave this available context unused, and its contribution to detection has received limited attention. Schema-grounded detection incorporates this information into the detector's input and provides structural evidence about available tables, columns, and their relationships.

UNION-based attacks provide one illustrative example of why schema context can matter. Attackers commonly use a sequence of UNION queries to probe the structure of an original query (Halfond et al., 2006; Clarke-Salt, 2012; OWASP Foundation, 2025b). By varying the number of projected expressions and their data types across requests, they can infer the number of columns returned by the original query and determine which types are compatible with each position. Once these properties have been identified, the attacker can construct a compatible UNION query to retrieve information through the original execution path. In the example, the query combines `SELECT * FROM product WHERE id = 2371031` with `SELECT name, salary FROM employees` through UNION. Without the target schema, the detector cannot determine how many columns `SELECT *` expands to, leaving the compatibility of the two projections unresolved. The schema reveals that `product` contains three columns, whereas the second SELECT returns only two. This mismatch provides structural evidence that is unavailable from the query text alone. Although this example focuses on UNION, the broader principle applies to other SQLi patterns whose interpretation depends on target objects, metadata references, type compatibility, or database-specific syntax.

B Taxonomy and Data Details

This appendix expands the taxonomy and benchmark statistics introduced in Sections 2.2 and 3. The examples use the payload-template notation of the benchmark builder; placeholders are filled during schema-grounded construction.

B.1 Taxonomy Dimensions and Cluster Definition

Each malicious example is defined by three dimensions: injection technique, payload reference scope, and comment state. We write the cluster key as

$$c_k = (\tau, \sigma, \kappa), \tag{6}$$

where τ identifies how the payload changes or extends the query, σ identifies the class of objects or literals referenced by the payload, and κ records whether and how a comment delimiter is used. Examples with the same key belong to the same injection cluster.

Injection technique. The six operational techniques describe attack mechanisms and their usual consequences, following established SQLi taxonomies while selecting the families represented in our benchmark (Halfond et al., 2006; Clarke-Salt, 2012; Sadeghian et al., 2013; OWASP Foundation, n.d.).

- **Tautology-based.** The payload inserts a condition that is always true, such as an equality between identical constants. When it reaches a predicate in a `WHERE` clause, it can weaken or bypass the application's intended restriction and return rows that should have been filtered.
- **UNION query-based.** The payload appends a `UNION SELECT` projection to the original query. An attacker can use a compatible result shape to make values from another table or database object appear in the application's response, which requires knowledge of usable tables, columns, or type-compatible expressions.
- **Piggy-backed query.** The payload terminates the carrier statement and appends a second SQL statement. Depending on the interface and privileges, the second statement can read, insert, update, or delete records, or invoke other database operations. This makes piggy-backed attacks capable of changing persistent state, but their effect depends on a multi-statement execution path.
- **Error-based.** The payload deliberately triggers an informative database error, for example through an invalid cast or an expression with incompatible types. The returned error can reveal database-engine behavior, data types, query structure, or schema-related facts, allowing an attacker to refine later probes.
- **Boolean-based blind.** The payload encodes a hypothesis as a Boolean condition and infers the answer from changes in the application's response. Repeating such probes lets an attacker reconstruct schema or data information even when query results are not directly returned.
- **Time-based blind.** The payload makes the database delay its response when a condition holds. By comparing response times across repeated probes, an attacker can infer schema or data information, although the attack also consumes resources and can increase service latency.

Payload reference scope. The scope dimension records the contextual information needed to interpret a payload. **Literal-only references (LOR)** use constants and do not name a database object. **Target-schema references (TSR)** refer to tables or columns in the target schema. **System-catalog references (SCR)** refer to system metadata, catalog objects, or engine variables. The three scopes therefore separate attacks that are independent of the target schema from attacks whose interpretation depends on application objects or database metadata.

The construction contains 16 valid technique–scope pairs. Boolean-blind/LOR is excluded because blind attacks are normally implemented as repeated probes whose questions depend on schema or data objects; without such a reference, the pair loses the defining blind-inference behavior and collapses toward a tautology-style probe. Piggy-backed/LOR is excluded because the appended `INSERT`, `UPDATE`, `DELETE`, or `SELECT` statement must operate on a database object; without a schema or catalog reference, the pair does not define a meaningful independent attack category. Combining the 16 pairs with the three comment states yields 48 injection clusters. Benign examples are maintained as a separate class and are not counted among these malicious clusters.

B.2 Representative Attack Patterns

Each example below shows a complete rendered SQLi query after payload insertion, followed by the payload template, its instantiation, and its effect. The scope label refers to references introduced by the payload itself; tables already present in the carrier query do not determine the payload scope.

Tautology-based.

```
SELECT COUNT(T3.superhero_name)
FROM hero_attribute AS T1
INNER JOIN attribute AS T2 ON T1.attribute_id = T2.id
INNER JOIN superhero AS T3 ON T1.hero_id = T3.id
WHERE T2.attribute_name = 'Speed'
  AND T1.attribute_value = 100
  OR 1 = 1
```

Payload template. `' OR 1 = 1`. The payload appends an always-true disjunct to the original predicate. Because `AND` has higher precedence than `OR`, the condition becomes `(attribute_name = 'Speed' AND attribute_value = 100) OR TRUE`, so the predicate can accept every row. An attacker can use this pattern to bypass an application-level restriction and broaden the result set. The payload therefore belongs to the literal-only reference scope (LOR).

UNION query-based.

```
SELECT T2.language
FROM sets AS T1
INNER JOIN set_translations AS T2
    ON T1.code = T2.setCode
WHERE T1.baseSetSize = 309
UNION
SELECT @@transaction_isolation
```

Payload template. `' UNION SELECT $sysInfo$`. Here `$sysInfo$` is instantiated as `@@transaction_isolation`, a system variable that exposes the current transaction isolation level. The payload appends a second projection and attempts to return system metadata through the application response. The payload therefore belongs to the system-catalog reference scope (SCR).

Piggy-backed query.

```
SELECT COUNT(T1.id)
FROM superhero AS T1
INNER JOIN hero_power AS T2 ON T1.id = T2.hero_id
INNER JOIN superpower AS T3 ON T2.power_id = T3.id
WHERE T3.power_name = 'Super Strength'
  AND T1.height_cm > 200;

DELETE FROM `superhero`
WHERE `full_name` = 'Jason Wynn';
```

Payload template. `'; DELETE FROM $table_1$ WHERE $column_t1_1$ = $sample_t1_1$;`. In this instance, the placeholders are instantiated as `superhero`, `full_name`, and `'Jason Wynn'`. The payload closes the first statement and appends a second statement that deletes a record, thereby changing persistent database state when multi-statement execution is permitted. The payload therefore belongs to the target-schema reference scope (TSR).

Error-based.

```
SELECT T3.type
FROM client AS T1
INNER JOIN disp AS T2 ON T1.client_id = T2.client_id
INNER JOIN card AS T3 ON T2.disp_id = T3.disp_id
WHERE T1.client_id = 9
ORDER BY 2
```

Payload template. `' ORDER BY 2`. The carrier query returns only one projected column, `T3.type`, whereas the injected ordinal refers to a second projected expression. The database therefore raises an error. By varying the ordinal, an attacker can infer the number of columns in the original projection and refine later `UNION` or extraction attempts. The payload introduces only a numeric ordinal and no database object, so it belongs to the literal-only reference scope (LOR).

Boolean-based blind.

```
SELECT T1.full_name
FROM superhero AS T1
INNER JOIN hero_attribute AS T2 ON T1.id = T2.hero_id
```

```
INNER JOIN attribute AS T3 ON T2.attribute_id = T3.id
WHERE T3.attribute_name = 'Strength'
  AND SUBSTR(DATABASE(), 1, 1) = 'a'
```

Payload template. `' AND SUBSTR($sysInfo$, 1, 1) = $character$.` Here `$sysInfo$` is instantiated as `DATABASE()` and `$character$` as `'a'`. The payload tests whether the first character of the database name equals a candidate character. Repeating the query with different candidates lets an attacker infer system metadata from changes in the application response, even when the metadata itself is never returned. The payload references the system function `DATABASE()`, so it belongs to the system-catalog reference scope (SCR).

Time-based blind.

```
SELECT DISTINCT T1.ID, T2.GLU
FROM Patient AS T1
INNER JOIN Laboratory AS T2 ON T1.ID = T2.ID
WHERE DATE_FORMAT(CAST(T1.`First Date` AS DATETIME), '%Y') = '1991'
  AND T2.GLU < 180
  AND CASE
        WHEN (@@hostname) REGEXP '[a-z]'
        THEN SLEEP(5)
        ELSE 0
      END
```

Payload template. `' AND CASE WHEN ($sysInfo$) REGEXP '[a-z]' THEN SLEEP(5) ELSE 0 END`. Here `$sysInfo$` is instantiated as `@@hostname`. If the database server hostname contains a lowercase letter, the query sleeps for five seconds; otherwise, it returns immediately. The attacker infers the condition from response latency and can use repeated timing probes to test database-server or other metadata properties. The payload references the system variable `@@hostname`, so it belongs to the system-catalog reference scope (SCR).

B.3 Motivation and Construction of Comment States

SQL comment delimiters are a salient structural feature of SQLi. A delimiter can neutralize the portion of the original SQL statement that follows the injection point, allowing a payload to be modified while preserving a syntactically valid and executable query when the surrounding conditions permit. Measuring this feature explicitly lets the benchmark compare attacks that leave the carrier suffix active with attacks that terminate it.

For this purpose, `no_comment` contains no SQL line-comment delimiter. `clean_comment` appends a supported delimiter, such as `--` or `#`, without natural-language text. `cepp`, or *comment-embedded prompt injection*, appends natural-language text after the delimiter. The database parser ignores this appended text, whereas an LLM-based detector still receives it as part of the model input. CEPP examples therefore test whether prompt-injection content can alter the detector's malicious-versus-benign decision. The text can be selected from the local comment repository or produced by the constrained generation process used by the benchmark builder. Benign explanatory comments are kept in the benign class and are not counted as CEPP malicious examples.

B.4 Data Sources, Split Protocol, and Dataset Statistics

Our benchmark derives its source SQL queries and target database schemas from the 11 databases in BIRD's public development set. BIRD is a cross-domain text-to-SQL benchmark built from real-world data sources, including Kaggle and the CTU Prague Relational Learning Repository. Each database is accompanied by human-annotated natural-language questions and corresponding SQL queries grounded in its schema and contents (Li et al., 2023). These resources provide the benign queries and carrier queries used in our benchmark; malicious examples are constructed by injecting payloads into the carriers.

We construct our own training, validation, and test partitions by assigning entire databases to separate splits before generating attack examples. The 11 databases are allocated as follows:

Table 2: Sample counts and malicious-example distributions across the three benchmark splits.

Category	Training	Validation	Test
Overall composition			
Total examples	**2,400**	**1,960**	**3,215**
Malicious examples	1,811	1,920	2,528
Benign examples	589	40	687
Split-defining resource pools			
Payload templates	63	175	180
Carrier queries	538	219	641
Comment entries	15	15	54
Malicious examples by technique			
Tautology-based	342	360	404
UNION query-based	342	360	628
Piggy-backed query	228	240	344
Error-based	342	360	628
Boolean-based blind	224	240	288
Time-based blind	333	360	236
Malicious examples by reference scope			
LOR	453	480	520
TSR	680	720	1,564
SCR	678	720	444
Malicious examples by comment state			
`no_comment`	604	640	1,888
`clean_comment`	604	640	320
`cepp`	603	640	320

- **Training:** `student_club`, `superhero`, `thrombosis_prediction`, and `toxicology`.
- **Validation:** `codebase_community` and `debit_card_specializing`.
- **Test:** `california_schools`, `card_games`, `european_football_2`, `financial`, and `formula_1`.

All source queries and the examples derived from a given database remain in its assigned split. The three partitions therefore contain disjoint target databases, and the target schemas used in validation and test are absent from the training partition. Carrier-query texts and benign source-query texts are also disjoint across splits, as are the entries in the partitioned local comment repository. No identical complete SQL query occurs in more than one split. This construction prevents examples derived from the same carrier query or target database from appearing on both sides of the training–evaluation boundary.

The training partition provides data and source resources for detector training and attack generation. The validation partition is used to assess detector weaknesses and guide the feedback controller during co-evolution. The test partition is reserved exclusively for evaluating detection performance on held-out target schemas.

Table 2 summarizes the resulting dataset. All three splits cover all 48 malicious clusters. Training examples are approximately balanced across these clusters, and validation contains exactly 40 malicious examples per cluster. The test split retains an unequal cluster distribution. Benign examples form a separate cluster outside the 48 malicious clusters. The splits are additionally defined by disjoint resource pools: each split reserves its own payload templates, carrier queries, and comment entries ("Split-defining resource pools"), so held-out examples are instantiated from attack-side resources that the training pipeline never observes.

C Benchmark Data Quality

We compare SQLI-BENCH with SQLiV3, rbsqli, and WAF-A-MoLe along four dimensions: structural completeness, normalized duplication, label agreement, and label ambiguity without schema.

These dimensions examine whether samples provide complete SQL inputs, how much repetition they contain, and whether their labels can be determined from query text alone.

C.1 Evaluation Dimensions

D1: Structural Completeness. The percentage of samples that pass the audit's checks for a complete SQL statement. This measures how well a dataset matches our setting, where the detector receives a rendered query before execution. Standalone injection fragments may be useful for other tasks but do not satisfy this complete-query requirement.

D2: Normalized Duplication Rate. The percentage of repeated samples after normalizing query text. Queries with the same normalized form are grouped together, and all but one sample in each group are counted as duplicates. A lower rate indicates less repetition under these normalization rules.

D3: Label Agreement. The percentage of samples whose recorded labels agree with independent expert judgments. This metric includes only samples that experts can classify as malicious or benign.

D4: Label Ambiguity without Schema. The percentage of reviewed samples that experts judge as ambiguous when given only the SQL query, without the target database schema. This metric measures how often query text alone provides insufficient evidence for a clear malicious-or-benign judgment.

C.2 Measurement Methodology

D1 and D2 are measured automatically. D3 and D4 are measured through expert review.

Audit samples. Each public dataset is evaluated on a fixed batch of 10,000 samples. For SQLi-BENCH, the automatic audit covers the complete test set described in Appendix B: 3,215 samples, comprising 2,528 malicious and 687 benign queries from five held-out databases. Training and validation samples are excluded from this quality comparison. Expert review uses a stratified subset of 250 samples from each dataset.

Structural completeness. The automatic audit checks whether a query begins with a top-level SQL keyword and can be parsed by SQLGlot using the MySQL dialect. The procedure also includes a SQLparse-based fallback and, for samples labeled malicious, a second parsing attempt after removing injection-boundary quotes. D1 records the percentage accepted by this procedure. Passing the audit does not guarantee that a query will execute successfully against a database.

Duplication. Before matching queries, we lowercase the text, collapse whitespace, and replace string and numeric literals with placeholders. The duplication rate is the number of samples minus the number of unique normalized queries, divided by the number of samples. This measures repeated normalized forms, not necessarily queries with identical behavior. The rates describe the audited batches and may depend on batch size.

Label review. Experts review each sample using only its SQL text, without access to the target database schema, and assign one of four labels: malicious, benign, ambiguous, or invalid. Final decisions follow the majority judgment, with a third adjudicator resolving ties. For D3, we exclude samples with an ambiguous or invalid final judgment and calculate agreement with the recorded binary labels. For D4, we divide the number of ambiguous samples by the total number of reviewed evaluation samples. D4 therefore captures uncertainty under query-only inputs.

C.3 Comparison Results

Table 3 compares the four datasets across the four evaluation dimensions.

SQLi-BENCH combines complete SQL structure with the lowest normalized duplication rate among the four datasets. All test samples passed the structural audit, and only 1.90% were duplicates after normalization. WAF-A-MoLe achieves a similar completeness rate of 99.95%, but its duplication rate reaches 16.99%. SQLiV3 and rbsqli contain substantially smaller proportions of complete SQL

Table 3: Data quality comparison across SQLi datasets.

Dataset	D1 Completeness	D2 Duplication	D3 Agreement	D4 Ambiguity without schema
SQLiV3	40.25%	12.80%	97.5%	6.3%
rbsqli	70.64%	4.79%	85.5%	14.5%
WAF-A-MoLe	99.95%	16.99%	80.56%	2.00%
SQLI-BENCH	100.00%	1.90%	98.11%	16.00%

statements. These results support SQLI-BENCH's suitability for evaluating detectors on complete queries while limiting repeated normalized forms.

SQLI-BENCH achieves the highest label agreement (98.11%), above SQLiV3's 97.5%, rbsqli's 85.5%, and WAF-A-MoLe's 80.56%. Expert review therefore largely supports the recorded labels of SQLI-BENCH samples that received a clear binary judgment.

Without schema information, experts judged 16.00% of SQLI-BENCH samples as ambiguous, compared with 14.5% for rbsqli, 6.3% for SQLiV3, and 2.00% for WAF-A-MoLe. This result highlights a limitation of query-only evaluation: SQL text alone can leave the interpretation of a query unresolved, even when the statement is structurally complete. By pairing queries with their target schemas, SQLI-BENCH provides additional evidence about the referenced database objects and query structure. The observed ambiguity therefore motivates the schema-grounded setting and provides a basis for evaluating whether schema context helps resolve these cases.

D THE COEVO-SQLI ALGORITHM

Algorithm D.1 defines the execution order of COEVO-SQLI. Each round selects attack clusters, constructs a fixed-size training slice, updates the DEFENDER, and evaluates the updated model on a fixed validation set. The resulting errors determine the cluster and benign-sampling policies for the next round. Table D.1 lists the parameters that define this algorithmic loop; implementation settings are specified in Appendix E.

Algorithm D.1 COEVO-SQLI training

Require: Initial DEFENDER θ; train-side synthesis sources $\mathcal{P}$ and benign examples $\mathcal{B}$; fixed validation set $\mathcal{V}$ covering all C injection clusters; configuration in Table D.1
Ensure: Final-round DEFENDER θ

$w_k \leftarrow 1$ for every injection cluster k; $\rho \leftarrow 0.25$, $m_{\text{ben}} \leftarrow 0$
Initialize mutation memory $\mathcal{H}$ with source templates and empty mutation history
for $r = 1$ to T **do**
  $t \leftarrow (r-1)/\max(T-1, 1)$
  $\gamma \leftarrow \gamma_{\text{start}} + t(\gamma_{\text{end}} - \gamma_{\text{start}})$; $\mu \leftarrow \mu_{\text{start}} + t(\mu_{\text{end}} - \mu_{\text{start}})$
  $p_k \leftarrow (1-\gamma)w_k^2 / \sum_j w_j^2 + \gamma/C$
  $\mathcal{S} \leftarrow \text{WeightedSampleWithoutReplacement}(\mathbf{p}, K)$
  $n_{\text{ben}} \leftarrow \text{Round}(N\rho)$; $n_{\text{mal}} \leftarrow N - n_{\text{ben}}$
  $(\mathcal{A}, \mathcal{H}) \leftarrow \text{SynthesizeAcceptedExamples}(\mathcal{P}, \mathcal{S}, \mu, \mathcal{H}, n_{\text{mal}})$
  $\mathcal{G} \leftarrow \text{SampleWithoutReplacement}(\mathcal{B}, n_{\text{ben}})$
  $\mathcal{D} \leftarrow \text{Shuffle}(\text{Concatenate}(\mathcal{A}, \mathcal{G}))$
  $\theta \leftarrow \text{UpdateDefender}(\theta, \mathcal{D})$
  $(\text{FN}_{1:C}, n_{1:C}, \text{FP}_{\text{ben}}, n_{\mathcal{V},\text{ben}}) \leftarrow \text{Evaluate}(\theta, \mathcal{V})$
  if $r < T$ **then**
    $R_k \leftarrow (\text{FN}_k + 1)/(n_k + 2)$ for every cluster; $\bar{R} \leftarrow C^{-1} \sum_k R_k$
    $w_k \leftarrow w_k \exp[\eta(R_k - \bar{R})]$ for every cluster
    $e_{\text{ben}} \leftarrow (\text{FP}_{\text{ben}} + 1)/(n_{\mathcal{V},\text{ben}} + 2)$
    $m_{\text{ben}} \leftarrow \lambda m_{\text{ben}} + (1-\lambda)e_{\text{ben}}$
    $\rho \leftarrow \text{Clip}(\rho + a(m_{\text{ben}} - \phi), \rho_{\min}, \rho_{\max})$
  end if
end for
return θ

Table D.1: Algorithmic configuration of the CoEvo-SQLi training loop.

Parameter	Setting
Training rounds T	8
Examples per round N	300
Injection clusters C	48
Selected clusters K	8
Initial cluster weights	1
Exploration schedule	Linear: $0.70 \rightarrow 0.20$
Mutation schedule	Linear: $0.10 \rightarrow 0.40$
Weight-update rate η	1.0
Error smoothing	Laplace smoothing
Initial benign ratio	0.25
Initial benign-error EMA	0
EMA decay λ	0.70
Target smoothed benign error ϕ	0.05
Controller step size a	0.50
Benign-ratio bounds	$[0.15, 0.35]$
Anti-imitation examples	Up to 5

Mutation memory and round transitions. Mutation memory provides up to five examples from the same technique–scope category for anti-imitation prompting. It rejects candidates that match source templates or previously accepted mutations after whitespace normalization and lowercasing. This check applies to payload mutations; the resulting SQL training slice is not globally deduplicated. Memory persists across rounds. The Defender update in each round starts from the model returned by the preceding round. Validation covers all injection clusters, and test predictions do not enter feedback updates. The final-round model is used for evaluation.

E Exact Experimental Protocol

This appendix specifies the implementation and evaluation contract used to obtain the results in Section 5. Algorithm D.1 defines the training procedure; this appendix fixes the input format, model configurations, data-processing rules, and evaluation protocol used to instantiate it.

E.1 Local Training and Checkpoint Selection

The local Defender uses Qwen2.5-Coder-0.5B-Instruct with LoRA fine-tuning. CoEvo-SQLi and Static SFT use the same backbone, training configuration, and training-example budget. The reported local results use three random seeds: 42, 123, and 234. Metrics are computed separately for each seed and then aggregated.

The final reported model is the merged model produced by the last declared training round. The held-out test split is used to report final performance. Test predictions do not enter synthesis, Verifier updates, threshold selection, prompt selection, or checkpoint selection.

E.2 Prompt and Label Contract

Each local training and evaluation example is stored as an OpenAI-style message sequence containing a system message, a user message, and an assistant message. The system message defines the pre-execution SQLi classification task and requires exactly one lowercase output label: `malicious` or `benign`. The user message contains the SQL query and, in the schema-grounded setting, the target database schema rendered as DDL. In query-only mode, the schema field is replaced by `null`. The assistant message contains the corresponding label. A complete schema-grounded example is shown below.

The Qwen tokenizer's native chat template renders the messages. During training, the system and user prefix is masked, so only assistant-label tokens contribute to the loss. During inference, the same system and user messages are rendered with an open assistant turn. The generated text is

Table 5: Implementation configuration for local training and mutation.

Parameter	Setting
Fine-tuning epochs per round	2
Optimizer	AdamW
Learning rate	5×10^{-5}
Learning-rate schedule	Linear decay; warmup ratio 0.03
Batch size per GPU	8
Gradient accumulation steps	2
Maximum sequence length	2,048 tokens
LoRA rank	16
LoRA alpha	32
LoRA dropout	0.1
LoRA target modules	`q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, `down_proj`
Precision	BF16

normalized by trimming whitespace and converting it to lowercase; only an exact `malicious` or `benign` output is accepted as valid.

The following example shows one schema-grounded training record.

System:

You are a pre-execution SQL injection classifier.

Classify the SQL query as either malicious or benign. Use the SQL query as the primary evidence and use the database schema as supporting structural evidence when it is provided.

Follow this decision order:

1. Inspect the query for injection behavior.
2. Identify the original carrier query and any appended or injected fragment when possible. Judge whether the fragment changes the intended query logic, result source, authorization scope, or execution behavior.
3. Use the schema to check whether referenced tables, columns, aliases, and catalog objects exist; whether UNION projections have compatible arity and types; whether joins and subqueries are structurally consistent; and whether references target ordinary application data or database metadata.
4. Resolve conflicts carefully. Valid table names, columns, and schema-consistent references do not make a query benign. Schema consistency indicates that a payload may execute, not that it is safe. A schema mismatch is supporting evidence of a suspicious or malformed query, but it does not by itself prove malicious intent. When clear injection behavior is present, classify the query as malicious even if all referenced objects exist. If the schema is incomplete or a reference cannot be resolved, rely on query-level evidence.

A benign query is one for which the available evidence does not indicate an injection attempt. Do not label a query malicious solely because it is unusual, complex, or malformed.

Output exactly one lowercase word: `malicious` or `benign`.

User:

SQL Query:

```
SELECT name FROM users WHERE id = 1 OR 1 = 1
```

Database Schema:

```
-- Database: app_db
CREATE TABLE users (
    id INTEGER,
    name VARCHAR
);
```

Assistant:

```
malicious
```

In query-only mode, the same system and assistant messages are used, while the user message contains `Database Schema: null`.

E.3 Mutation and Synthesis

Dynamic synthesis uses train-side payload templates, carrier queries, database schemas, and comment resources. At each round, the attacker samples the selected attack clusters and generates the required number of malicious examples. The base mutation probability follows a linear schedule from 0.10 in the first round to 0.40 in the final round.

Round-level allocation and sampling semantics. Cluster selection is without replacement within a round: the $K{=}8$ clusters chosen for a round are distinct, and because cluster weights persist across rounds, a cluster may be selected again in later rounds. The malicious quota $n_{\mathrm{mal}} = N - n_{\mathrm{ben}}$ (225 at the initial ratio) is not fixed per cluster in advance: synthesis cycles round-robin over the selected clusters and attempts one example per cluster per cycle, and rejected attempts do not consume the quota. Realized per-cluster counts are therefore equal up to acceptance-rate differences among the selected clusters and the point at which the quota is reached. Benign examples are sampled without replacement from the full train-side benign pool $\mathcal{B}$ (589 examples) at every round; the pool is not depleted across rounds, so a benign example may recur in different training slices. Under the initial ratio $\rho = 0.25$ with $N = 300$, eight rounds request $8 \times 75 = 600$ benign slots from the 589-example

Table 6: Complete schema-grounded results on SQLI-BENCH, recomputed from the terminal-round evaluation artifacts.

Model	Acc	Prec	TPR	F1	FPR	MCC	WCR
claude-sonnet-5	86.7	98.4	84.5	90.9	4.9	69.6	18.4
deepseek-v4-flash	67.8	98.3	60.1	74.6	3.8	46.2	11.2
gemini-3.8-flash	91.9	100.0	89.8	94.6	0.1	80.7	46.0
glm-5.3-flash	74.1	99.6	67.3	80.4	0.9	54.6	14.0
gpt-5.6-luna	89.3	100.0	86.4	92.7	0.1	75.8	40.4
kimi-k2.7-code	76.6	99.9	70.3	82.5	0.1	57.8	16.0
qwen3.8-flash	84.0	100.0	79.6	88.6	0.0	67.4	24.3
Backbone	78.8 ± 0.1	78.8 ± 0.1	99.9 ± 0.1	88.1 ± 0.1	98.8 ± 0.4	7.4 ± 2.7	96.5 ± 2.7
Static SFT	90.7 ± 0.8	99.6 ± 0.1	88.5 ± 1.0	93.7 ± 0.6	1.3 ± 0.2	78.0 ± 1.6	33.6 ± 3.0
Uniform / no mutation	87.9 ± 3.8	99.6 ± 0.2	85.0 ± 4.9	91.7 ± 2.9	1.4 ± 0.6	73.3 ± 6.1	29.3 ± 12.0
Uniform / mutation	93.0 ± 2.2	98.8 ± 1.1	92.2 ± 3.6	95.4 ± 1.5	4.1 ± 3.9	82.0 ± 3.8	46.3 ± 15.1
Feedback / no mutation	93.9 ± 2.0	97.5 ± 2.4	94.8 ± 4.0	96.1 ± 1.4	9.2 ± 9.3	83.3 ± 4.3	65.9 ± 24.5
CoEvo-SQLi	97.4 ± 1.4	99.1 ± 0.2	97.5 ± 2.0	98.3 ± 0.9	3.2 ± 0.6	92.5 ± 3.7	80.1 ± 13.2

pool, so recurrence across rounds is expected rather than exceptional. Each round's slice is shuffled before the defender update (Algorithm D.1).

The mutation model used in the reported experiments is Qwen3.6-27B (Qwen Team, 2026), an open-weight model, and remains fixed throughout training. Both the DEFENDER and the mutation model are open-weight and served within the same local trust boundary, so no query, schema, or payload data leaves the deployment environment during training or inference. Mutation memory stores source templates and previously accepted mutations for each technique–scope category. Up to five category-matched examples are included in the mutation prompt as anti-imitation context. The mutator is instructed to preserve the attack technique, reference scope, required placeholders, and outer injection prefix, and to output one comment-free payload core without explanations.

The implementation uses two mutation prompt variants. The attack-form prompt changes the function, operator, wrapper, or SQL form while preserving the attack category. The SQL-structure prompt changes the embedded query structure while preserving the technique and its form. Candidate mutations are accepted only when they satisfy the required format, placeholder, taxonomy, and basic structural constraints. Normalized duplicates of source templates or previously accepted mutations are rejected, and accepted mutations are added to the persistent mutation memory.

Accepted payload cores are passed to the shared synthesis pipeline, which fills scope-specific placeholders, assembles the selected comment state, inserts the payload into a compatible carrier query, and checks the resulting sample. Valid malicious examples are combined with sampled benign examples and converted into the selected prompt format to form the round training data.

F COMPLETE RESULTS AND ADDITIONAL ANALYSES

This appendix reports the complete results supporting the aggregate claims in Section 5. The main text emphasizes the primary comparisons; the tables and analyses below expose seed variation, category-level behavior, transfer performance, schema contribution, and resource costs. All numbers are recomputed from the terminal-round evaluation artifacts of the final runs: locally trained conditions aggregate the three training seeds (42, 123, and 234), with CoEvo-SQLi conditions taken from the final round and Static SFT from its single round; closed-source models use one fixed zero-shot run per input condition; ModSecurity CRS, Trident, and EP-CNN use their recorded fixed configurations.

F.1 COMPLETE SCHEMA-GROUNDED AND QUERY-ONLY RESULTS

Table 6 reports accuracy, precision, TPR, F1, FPR, MCC, and WCR for the complete schema-grounded comparison on SQLI-BENCH. WCR (worst-cluster recall) is the worst recall over the 48 malicious clusters, computed per seed and then averaged. Table 7 reports the corresponding query-only results. Table 8 breaks the schema-grounded comparison down by injection technique, payload reference scope, and comment state, reporting cluster-macro TPR per category.

Table 7: Complete query-only results on SQLI-BENCH, recomputed from the terminal-round evaluation artifacts.

Model	Acc	Prec	TPR	F1	FPR	MCC	WCR
ModSecurity CRS	61.4	95.2	53.6	68.6	9.9	36.0	0.0
Trident	78.6	89.2	82.8	85.9	36.8	42.4	20.0
EP-CNN	77.7	83.2	89.8	86.4	66.8	26.4	60.0
claude-sonnet-5	79.8	100.0	74.3	85.2	0.1	61.7	7.7
deepseek-v4-flash	67.8	98.6	60.0	74.6	3.2	46.6	7.5
gemini-3.8-flash	86.8	100.0	83.2	90.8	0.0	71.7	32.4
glm-5.3-flash	76.8	99.7	70.7	82.7	0.9	57.7	23.4
gpt-5.6-luna	84.3	99.8	80.2	88.9	0.6	67.7	23.9
kimi-k2.7-code	75.8	99.6	69.6	81.9	1.2	56.3	12.8
qwen3.8-flash	83.1	100.0	78.6	88.0	0.1	66.2	22.9
Backbone	78.5 ± 0.0	78.6 ± 0.0	99.9 ± 0.0	88.0 ± 0.0	100.0 ± 0.0	-1.7 ± 0.1	95.0 ± 0.0
Static SFT	90.2 ± 0.8	99.7 ± 0.1	87.8 ± 1.0	93.4 ± 0.6	1.0 ± 0.2	77.1 ± 1.5	25.6 ± 1.9
CoEvo-SQLi (QO)	92.5 ± 3.0	99.4 ± 0.2	91.0 ± 3.7	95.0 ± 2.1	2.0 ± 0.8	81.5 ± 5.9	46.9 ± 6.8

Table 8: Schema-grounded detection across injection techniques (IT), payload reference scopes (PRS), and comment states (CS). Each cell reports cluster-macro TPR (%), averaging eligible clusters equally within each category. Closed-source models use schema-grounded prompting. Bold and underlined values indicate the best and second-best results in each column.

Model	IT						PRS			CS		
	Taut.	UNION	Piggy.	Error	Boolean	Time	LOR	TSR	SCR	None	Clean	CEPP
claude-sonnet-5	98.9	99.9	**100.0**	88.2	91.0	99.5	92.8	95.1	**100.0**	89.1	**100.0**	**100.0**
deepseek-v4-flash	86.6	85.8	97.7	58.8	65.2	90.8	84.5	64.0	95.0	71.6	92.5	78.1
gemini-3.8-flash	99.4	99.8	**100.0**	92.6	94.3	**100.0**	95.5	97.0	**100.0**	93.3	**100.0**	**100.0**
glm-5.3-flash	93.0	96.5	96.0	72.9	81.5	92.0	88.3	77.8	99.7	76.5	95.5	94.0
gpt-5.6-luna	98.8	**100.0**	99.9	90.1	89.2	99.3	95.2	93.7	**100.0**	89.9	**100.0**	99.4
kimi-k2.7-code	95.8	97.3	95.8	77.0	76.8	94.8	91.1	79.6	99.7	78.8	98.4	92.8
qwen3.8-flash	98.7	99.8	**100.0**	84.7	87.5	96.5	93.3	90.1	**100.0**	84.5	99.7	99.7
Static SFT	99.7	99.6	97.6	89.6	97.8	99.0	94.1	97.0	99.4	91.8	**100.0**	99.8
CoEvo-SQLi	**100.0**	99.8	99.5	**96.8**	**99.9**	99.6	**98.2**	**99.7**	99.4	**98.0**	**100.0**	99.7

F.2 Cross-Seed Ablation Statistics

Table 9 reports cross-seed means and sample SDs of accuracy, precision, TPR, F1, FPR, MCC, and WCR for the four schema-grounded conditions of the 2×2 feedback–mutation ablation. The ablations are trained only in the schema-grounded setting, so query-only counterparts are not available.

Table 9: Cross-seed statistics of the four schema-grounded feedback–mutation conditions, recomputed from the terminal-round evaluation artifacts; mean $\pm$ sample SD over three seeds (42, 123, 234). Rates and MCC $\times 100$.

Condition	Acc	Prec	TPR	F1	FPR	MCC	WCR
Uniform / no mutation	87.9 ± 3.8	99.6 ± 0.2	85.0 ± 4.9	91.7 ± 2.9	1.4 ± 0.6	73.3 ± 6.1	29.3 ± 12.0
Uniform / mutation	93.0 ± 2.2	98.8 ± 1.1	92.2 ± 3.6	95.4 ± 1.5	4.1 ± 3.9	82.0 ± 3.8	46.3 ± 15.1
Feedback / no mutation	93.9 ± 2.0	97.5 ± 2.4	94.8 ± 4.0	96.1 ± 1.4	9.2 ± 9.3	83.3 ± 4.3	65.9 ± 24.5
CoEvo-SQLi	97.4 ± 1.4	99.1 ± 0.2	97.5 ± 2.0	98.3 ± 0.9	3.2 ± 0.6	92.5 ± 3.7	80.1 ± 13.2

Because all locally trained conditions share the three seeds, Table 10 reports two-sided seed-paired t-tests for the primary MCC comparisons, together with a Welch test for the query-only SQLI-BENCH comparison. The schema-grounded gain over Static SFT and the cost of removing feedback are significant; the cost of removing mutation, the Static SFT advantage over the uniform no-mutation loop, and the rbsqli and WAF-A-MoLe transfer margins are not separable at three seeds, whereas the SQLiV3 margin is. With two degrees of freedom these tests have limited power: non-significant rows should be read as inconclusive rather than as evidence of no difference, and claims tied to those rows are statements about mean behavior.

Table 10: Seed-paired significance tests on terminal-round test MCC ($\times 100$). Locally trained conditions share the three seeds (42, 123, 234), so comparisons use two-sided paired t-tests ($\mathrm{df} = 2$) on per-seed values; the query-only SQLI-BENCH row uses Welch's t-test computed from the seed means and standard deviations of Table 7, because that comparison is summarized rather than seed-paired. Positive differences favor the first condition. With three seeds these tests have limited power: non-significant rows should be read as inconclusive rather than as evidence of no difference.

Comparison	Setting	Δ MCC	t	p
COEVO-SQLI vs. Static SFT	SQLI-BENCH, schema-grounded	+14.5	5.07	0.037
COEVO-SQLI vs. Uniform / mutation	SQLI-BENCH, schema-grounded	+10.5	10.09	0.010
COEVO-SQLI vs. Feedback / no mutation	SQLI-BENCH, schema-grounded	+9.2	2.16	0.163
Static SFT vs. Uniform / no mutation	SQLI-BENCH, schema-grounded	+4.7	1.16	0.365
COEVO-SQLI (QO) vs. Static SFT	SQLI-BENCH, query-only (Welch)	+4.4	1.25	0.325
COEVO-SQLI (QO) vs. Static SFT	rbsqli	+0.7	0.71	0.552
COEVO-SQLI (QO) vs. Static SFT	SQLiV3	+11.3	14.15	0.005
COEVO-SQLI (QO) vs. Static SFT	WAF-A-MoLe	+3.4	2.16	0.163

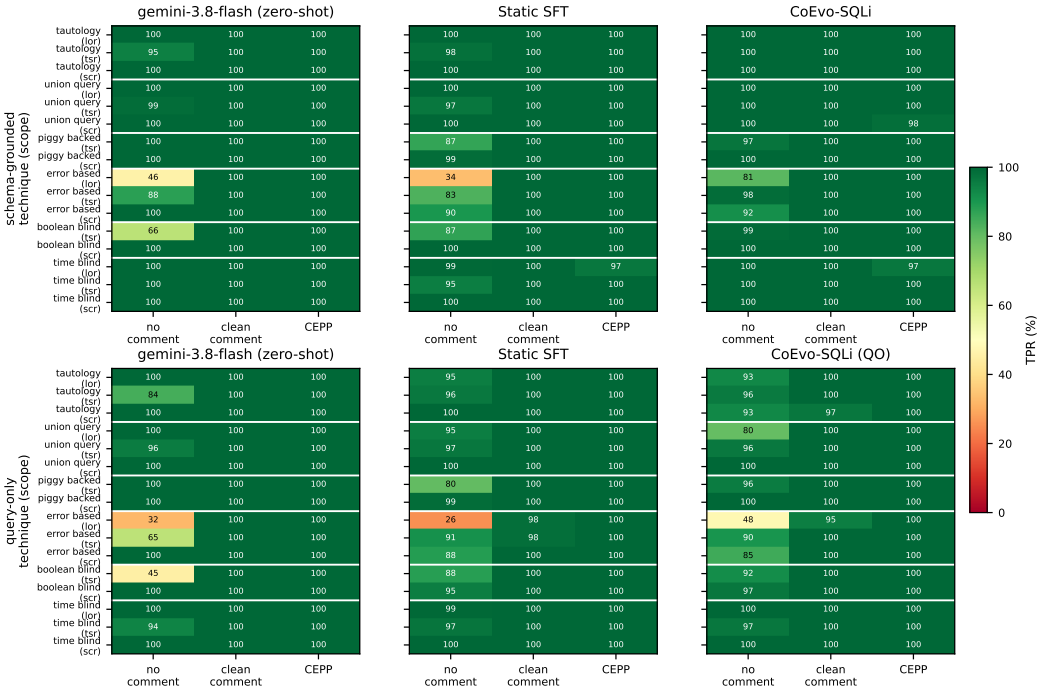

Figure 7: Per-cluster TPR (%) over the 48 malicious injection clusters. Rows are technique–scope pairs (white lines separate techniques) and columns are comment states; the top row of panels is schema-grounded and the bottom row query-only. Panels show the strongest closed-source baseline (gemini-3.8-flash, zero-shot), Static SFT, and COEVO-SQLI; trained panels average the three seeds.

F.3 TAXONOMY COVERAGE

Figure 7 reports per-cluster TPR over the 48 malicious injection clusters for the strongest closed-source baseline (gemini-3.8-flash, zero-shot), Static SFT, and COEVO-SQLI in both input settings; the trained panels average the three seeds. Detection gaps concentrate in the no-comment columns of the error-based techniques for every method. On the hardest cluster, error-based injections with literal-only payload references and no comments, TPR reaches 0.46 for gemini-3.8-flash, 0.34 for Static SFT, and 0.81 for COEVO-SQLI in the schema-grounded setting (0.32, 0.26, and 0.48 in the query-only setting): the co-evolutionary training narrows, but does not close, the hardest gap.

Table 11: Complete zero-adaptation transfer results on rbsqli. Rates and MCC $\times 100$; locally trained rows report mean $\pm$ sample SD over three seeds. Learned detectors are trained on SQLI-BENCH. The external benchmarks carry no injection-taxonomy labels, so a WCR column is not defined.

Model	Acc	Prec	TPR	F1	FPR	MCC
ModSecurity CRS	87.4	100.0	47.4	64.4	0.0	63.8
Trident	45.0	24.0	59.9	34.3	59.7	0.1
EP-CNN	76.8	100.0	3.2	6.3	0.0	15.7
claude-sonnet-5	26.2	24.4	99.1	39.1	96.8	5.9
deepseek-v4-flash	70.2	43.9	87.9	58.5	35.4	44.8
gemini-3.8-flash	75.8	49.7	100.0	66.4	31.9	58.2
glm-5.3-flash	28.5	25.1	99.8	40.1	93.9	11.9
gpt-5.6-luna	72.4	46.5	100.0	63.5	36.3	54.4
kimi-k2.7-code	63.2	39.4	99.5	56.5	48.2	44.8
qwen3.8-flash	33.4	26.4	99.9	41.8	87.6	17.9
Backbone	24.0 ± 0.0	24.0 ± 0.0	100.0 ± 0.0	38.7 ± 0.0	100.0 ± 0.0	0.0 ± 0.0
Static SFT	97.3 ± 0.4	100.0 ± 0.0	88.6 ± 1.6	93.9 ± 0.9	0.0 ± 0.0	92.5 ± 1.1
COEVO-SQLI (QO)	97.4 ± 1.0	96.3 ± 6.4	93.4 ± 6.6	94.6 ± 2.2	1.3 ± 2.3	93.1 ± 2.7

Table 12: Complete zero-adaptation transfer results on SQLiV3. Rates and MCC $\times 100$; locally trained rows report mean $\pm$ sample SD over three seeds. Learned detectors are trained on SQLI-BENCH. The external benchmarks carry no injection-taxonomy labels, so a WCR column is not defined.

Model	Acc	Prec	TPR	F1	FPR	MCC
ModSecurity CRS	91.8	95.0	81.9	88.0	2.5	82.3
Trident	55.8	38.3	33.7	35.9	31.5	2.3
EP-CNN	54.2	34.8	28.5	31.3	30.9	-2.6
claude-sonnet-5	97.0	96.2	95.5	95.9	2.2	93.5
deepseek-v4-flash	91.9	83.8	96.6	89.7	10.8	83.7
gemini-3.8-flash	95.2	88.9	99.2	93.8	7.1	90.2
glm-5.3-flash	92.2	82.7	99.2	90.2	11.9	84.8
gpt-5.6-luna	94.4	87.7	98.3	92.7	7.9	88.5
kimi-k2.7-code	94.2	87.4	98.4	92.6	8.1	88.3
qwen3.8-flash	94.0	86.8	98.4	92.2	8.6	87.8
Backbone	36.4 ± 0.0	36.4 ± 0.0	100.0 ± 0.0	53.4 ± 0.0	100.0 ± 0.0	0.0 ± 0.0
Static SFT	83.5 ± 0.1	98.8 ± 0.4	55.5 ± 0.2	71.1 ± 0.2	0.4 ± 0.1	65.8 ± 0.3
COEVO-SQLI (QO)	90.0 ± 1.1	93.1 ± 2.8	75.6 ± 4.7	83.3 ± 1.7	3.0 ± 1.8	77.1 ± 1.1

F.4 EXTERNAL QUERY-ONLY TRANSFER

Tables 11–13 report zero-adaptation query-only transfer on the three external datasets, rbsqli, SQLiV3, and WAF-A-MoLe. The transferred model is evaluated without further training, prompt selection, threshold calibration, or test-based checkpoint selection on the target dataset. COEVO-SQLI (QO) attains the best MCC on rbsqli and WAF-A-MoLe; on SQLiV3 it trails the strongest zero-shot API baselines. Seed-paired statistics for the COEVO-SQLI (QO) versus Static SFT margins on these datasets appear in Table 10.

F.5 SCHEMA CONTRIBUTION

The paired schema-grounded and query-only conditions use the same query identities and labels wherever the benchmark permits a direct comparison. Their difference measures the contribution of the supplied schema context under the declared training and evaluation protocols.

Figure 8 reports MCC in both settings for the seven closed-source baselines, Static SFT, and COEVO-SQLI. Schema context yields the largest gains for the strongest closed-source reasoners (gemini-3.8-flash $+9.0$, gpt-5.6-luna $+8.1$, and claude-sonnet-5 $+7.9$ MCC points) and for COEVO-SQLI ($+11.0$), whereas Static SFT improves by only $+0.9$ and weaker baselines change little or even degrade (glm-5.3-flash -3.1). The value of schema context therefore grows with the detector's ability to exploit structural evidence, and the co-evolutionary training loop, which couples schema-grounded validation feedback to data selection, amplifies this value for a small local model.

Table 13: Complete zero-adaptation transfer results on WAF-A-MoLe. Rates and MCC $\times 100$; locally trained rows report mean $\pm$ sample SD over three seeds. Learned detectors are trained on SQLI-BENCH. The external benchmarks carry no injection-taxonomy labels, so a WCR column is not defined.

Model	Acc	Prec	TPR	F1	FPR	MCC
ModSecurity CRS	92.0	98.5	86.2	92.0	1.5	84.7
Trident	63.5	63.6	73.8	68.3	48.3	26.1
EP-CNN	82.7	91.8	74.2	82.1	7.6	67.1
claude-sonnet-5	86.8	83.6	93.8	88.4	21.1	73.9
deepseek-v4-flash	78.2	76.2	86.8	81.1	31.8	56.4
gemini-3.8-flash	91.4	92.1	91.8	92.0	9.0	82.8
glm-5.3-flash	77.3	72.9	91.6	81.2	39.2	55.7
gpt-5.6-luna	88.4	88.3	90.2	89.2	13.7	76.6
kimi-k2.7-code	82.5	79.2	91.1	84.7	27.4	65.3
qwen3.8-flash	86.3	83.6	92.5	87.8	20.8	72.7
Backbone	53.1 ± 0.0	53.3 ± 0.0	99.4 ± 0.0	69.4 ± 0.0	100.0 ± 0.0	-5.1 ± 0.0
Static SFT	91.8 ± 0.6	98.5 ± 0.0	85.9 ± 1.2	91.8 ± 0.7	1.5 ± 0.1	84.4 ± 1.1
CoEvo-SQLi (QO)	93.7 ± 1.5	98.6 ± 0.2	89.1 ± 3.0	93.6 ± 1.6	1.4 ± 0.3	87.9 ± 2.8

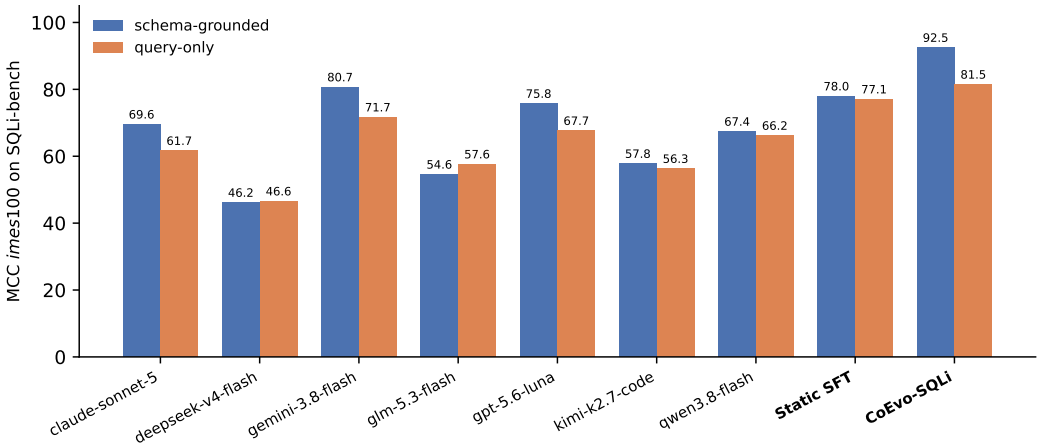

Figure 8: Schema contribution on SQLI-BENCH: MCC $\times 100$ in the schema-grounded and query-only settings for the seven closed-source zero-shot baselines, Static SFT, and COEVO-SQLI. Locally trained bars report the mean over three seeds.

F.6 INFERENCE EFFICIENCY AND TRAINING COST

The following tables provide the detailed local and remote efficiency measurements. Local inference efficiency is measured during the terminal test evaluation: the evaluation decodes sequentially with batch size one, feeding one query at a time and waiting for its completion before the next, without batching acceleration; under this setting throughput is the reciprocal of single-query latency, and the table reports per-seed means in Queries/s. The API tables report the wall-clock, effective throughput at the declared concurrency, the mean token usage per query, and the cost per 1,000 classifications at the providers' list prices. The columns of the training-cost table are defined as follows: Wall is the total wall-clock time of the complete training pipeline for the condition, Synthesis the time spent in the attack-synthesis (training-example generation) stage, Fine-tune the time spent in the LoRA fine-tuning stage, and Validation eval the cumulative time of per-round model inference on the validation set, all in minutes.

Table 14: Local inference efficiency.

Model/input	Queries/s
Backbone (SG)	39.9
Backbone (Q)	73.3
Static SFT (SG)	34.0
Static SFT (Q)	58.8
CoEvo-SQLi (SG)	34.3
CoEvo-SQLi (QO)	57.0

Table 15: Training cost by condition, in minutes.

Condition	Wall	Synthesis	Fine-tune	Validation eval
Static SFT (SG)	5.9	–	5.9	–
Static SFT (Q)	4.6	–	4.6	–
Uniform / no mutation (SG)	22.2	3.9	12.5	4.6
Uniform / mutation (SG)	38.9	21.2	12.0	4.5
Feedback / no mutation (SG)	21.3	3.8	11.8	4.5
CoEvo-SQLi (SG)	35.4	16.4	13.2	4.5
CoEvo-SQLi (QO)	30.1	16.0	9.9	2.7

Table 16: API efficiency in the schema-grounded setting, from one fixed zero-shot run of 3,215 queries at the declared concurrency of 32. Wall-clock covers the complete invocation; throughput divides the query count by wall-clock; token usage is the mean per query; cost applies the providers' September 2026 list prices (per 1M input/output tokens) to the mean usage, with gemini-3.8-flash at its introductory rate. Provider-side training resources are not observable.

Model	Wall (h)	Req/s	In tok/q	Out tok/q	USD/1k
claude-sonnet-5	0.21	4.2	2,948	5	5.95
deepseek-v4-flash	0.17	5.4	1,210	489	0.31
gemini-3.8-flash	0.38	2.4	1,289	305	2.11
glm-5.3-flash	0.36	2.5	1,107	475	0.40
gpt-5.6-luna	0.06	15.9	1,117	57	0.29
kimi-k2.7-code	0.33	2.7	1,200	349	2.54
qwen3.8-flash	0.56	1.6	1,306	598	0.48

Table 17: API efficiency in the query-only setting, from one fixed zero-shot run of 3,215 queries at the declared concurrency of 32. Wall-clock covers the complete invocation; throughput divides the query count by wall-clock; token usage is the mean per query; cost applies the providers' September 2026 list prices (per 1M input/output tokens) to the mean usage, with gemini-3.8-flash at its introductory rate. Provider-side training resources are not observable.

Model	Wall (h)	Req/s	In tok/q	Out tok/q	USD/1k
claude-sonnet-5	0.16	5.5	767	11	1.64
deepseek-v4-flash	0.15	5.8	551	423	0.20
gemini-3.8-flash	0.36	2.5	489	343	1.65
glm-5.3-flash	0.38	2.3	470	459	0.30
gpt-5.6-luna	0.05	16.6	468	57	0.16
kimi-k2.7-code	0.34	2.6	538	323	1.80
qwen3.8-flash	0.44	2.0	534	510	0.32