

Complete or Abstain? Aligning SQL Completion with User Preferences

Xintong Hu^{1*}, Defeng Xie^{3*}, Yuxiang Wang², Chunyou Li³, Haining Xie³,
Hao Liu⁴, Haozhe Feng³, Yang Li³, Danqing Huang³, Debiao He²,
Xiao Yan⁵, Jiawei Jiang^{2,6†}, Peng Chen^{3†}

¹School of Cyber Science and Engineering, Wuhan University,

²School of Computer Science, Wuhan University, ³Tencent Inc.,

⁴School of Mathematics, Peking University, ⁵Institute for Math and AI, Wuhan University,

⁶School of Computer Science, Central China Normal University

{xintong.hu, nai.yxwang, hedebiao, yanxiaosunny, jiawei.jiang}@whu.edu.cn liuhao_2002@stu.pku.edu.cn
{milesxie, chunyouli, hainingxie, aidenhzfeng, thomasyngli, daisyqhuang, pengchen}@tencent.com

Abstract

SQL code completion boosts developer productivity by predicting missing snippets from editing SQL contexts. However, existing solutions yield low acceptance since they cannot abstain when no suggestion is needed and lack SQL-specific training. To address both limitations, we introduce PACE-SQL, a preference-aligned framework that improves SQL completion accuracy and user preference alignment. We build two datasets with completion and abstention samples: WeSQL-1.5M from real-world interaction logs to capture authentic user preferences, and OpenSQL-FIM from public SQL corpora. We adopt continual pretraining and supervised fine-tuning to improve completion quality, and design a reward function in the reinforcement learning to enhance alignment with user preferences. Experiments show PACE-SQL improves Exact Match and Edit Similarity scores by 9.67% and 7.52% over the best baseline on WeSQL-1.5M and raises online user acceptance rate from 5% to 29%.

1 Introduction

The advanced code comprehension and generation capabilities of large language models (LLMs) enable code completion—automatically predicting missing code from the current editing context (OpenAI et al., 2024; Hou et al., 2024; Jiang et al., 2026). It is reported that over 70% of developers use code completion tools in daily coding (Wang et al., 2023), driving the emergence of AI-powered coding assistants such as GitHub Copilot¹ and Cursor².

Mainstream code completion models such as Qwen-Coder (Hui et al., 2024) and DeepSeek-

Coder (Guo et al., 2024) focus on general programming languages like Python, C, and Java, achieving strong performance on benchmarks such as HumanEval-FIM (Bavarian et al., 2022) and CrossCodeEval (Ding et al., 2023). These models typically adopt the Fill-in-the-Middle (FIM) paradigm (Bavarian et al., 2022), which trains models to generate missing code spans from surrounding context. However, their performance degrades on SQL completion tasks. SQL is the standard language for data querying, transformation, and modeling in enterprise data platforms (Shi et al., 2025). Therefore, improving SQL code completion is crucial for enterprise development.

We identify three key limitations when deploying existing methods in SQL editing scenarios:

- **Over-Suggestion.** Recent studies (Bakal et al., 2025; Prather et al., 2023) highlight over-suggestion as a prevalent issue in code completion, suggesting that an ideal model should provide accurate suggestions when needed and abstain otherwise. In contrast, standard generation objectives assume a non-empty target for every FIM sample. This encourages models to generate suggestions whenever completion is triggered, even when users do not need assistance.
- **Random FIM sampling.** In real IDE scenarios, completions are triggered when users pause during editing, e.g., after a GROUP BY, or within an unfinished join condition. Existing methods instead build FIM samples by randomly masking spans in completed files, mismatching the positions where completions are actually triggered in IDEs. This mismatch degrades the model’s completion quality in practice.
- **Scarce SQL datasets.** Current corpora and benchmarks predominantly target general programming languages, lacking high-quality SQL resources. Existing SQL data lacks both scale and user preference signals, such as when to

*Both authors contributed equally to this research.

†Corresponding authors.

This work was done during Xintong Hu’s internship at Tencent TEG Data Computing Platform Department.

¹<https://github.com/features/copilot>

²<https://cursor.com/>

complete or abstain. This data scarcity directly limits model performance on SQL completion.

To address these limitations, we propose **PACE-SQL** (Preference-Aligned Code completion framEwork for SQL), which jointly optimizes completion quality and preference alignment. Our key insight is that real-world IDE interaction logs reveal users’ genuine completion preferences by recording where completions are triggered, whether users adopt the suggestions, and how the code evolves afterward. We therefore mine these logs to construct preference-aware training data. To bridge the static FIM mismatch, we derive *completion-required samples* by comparing the code at each trigger position between the trigger time and later editing states, using the newly inserted span as the ground truth. For over-suggestion, we mine *abstention samples*, which target empty strings, from ignored suggestions with no subsequent edit at triggered positions. To alleviate SQL resource scarcity, we construct **WeSQL-1.5M** from interaction logs, capturing user preference signals. We additionally build **OpenSQL-FIM** from public SQL corpora by constructing FIM samples with abstention scenarios.

PACE-SQL trains in three stages. Continual Pre-training (CPT) and Supervised Fine-Tuning (SFT) internalize SQL-domain knowledge and completion quality. In the Reinforcement Learning (RL), we design a reward function that jointly rewards correct generation, user adoption, and conservative abstention, thereby preserving completion quality while suppressing over-suggestion.

We evaluate PACE-SQL on WeSQL-1.5M and OpenSQL-FIM. Compared with general LLMs, code LLMs, and external filters, PACE-SQL improves Exact Match (EM) and Edit Similarity (ES) over the best baseline by 9.67% and 7.52% on WeSQL-1.5M. Ablation studies confirm the contribution of each training stage, abstention data, and the reward function. PACE-SQL increases the online user acceptance rate from 5% to 29%.

We summarize our contributions as follows:

- We propose PACE-SQL, a preference-aligned SQL completion framework that integrates CPT, SFT, and RL to enhance completion quality and suppress noisy suggestions.
- We design a unified RL reward function that combines correct generation, user adoption, and abstention, enabling the model to learn when to complete and when to stay silent.
- We construct two SQL completion datasets: **WeSQL-1.5M**, mined from real-world user logs with 1.5 million samples, and **OpenSQL-FIM**, derived from public SQL corpora. Both datasets cover completion and abstention scenarios.³
- We conduct extensive experiments to validate PACE-SQL, with the online acceptance rate confirming its alignment with user preferences.

2 SQL Fill-in-the-Middle Formulation

Task. Given a complete SQL sequence x , the Fill-in-the-Middle (FIM) task partitions it into a prefix p , a middle segment m , and a suffix s , i.e., $x = (p, m, s)$. The goal is to generate the missing segment m conditioned on the surrounding context, formulated as $P(m \mid p, s)$.

Format. For FIM with LLMs, the three segments are serialized with sentinel tokens into the Prefix-Suffix-Middle (PSM) format: `<PRE> p <SUF> s <MID> m`.

Due to the page limit, we provide a detailed discussion of related work in Appendix A. In short, unlike prior work that relies on external filter models to suppress suggestions (Sun et al., 2024; Mozannar et al., 2024; de Moor et al., 2026), we model abstention as an internal generation behavior, removing the need for auxiliary modules.

3 WeSQL-1.5M Dataset Construction

3.1 SQL Corpus Construction

We collect raw SQL submission scripts from Tencent WeData platform, an enterprise data warehouse and governance platform where SQL is the dominant language among over 8.7K users and the system serves over 30K SQL completion requests per day. However, raw SQL data often contains syntax errors and redundant query templates. Therefore, we apply length filtering, deduplication, and syntax validation to construct a continual pre-training corpus $\mathcal{D}_{\text{CPT}}$. Detailed procedures are provided in Appendix G.

3.2 Mining Ground Truth for FIM Tasks

Conventional FIM data construction randomly splits code sequences into prefix, middle, and suffix triples, ignoring real user completion behaviors. However, online interaction logs record continuous editing trajectories and adoption of the gener-

³For privacy, only OpenSQL-FIM is publicly released. It is available at <https://github.com/DefengXie/PACE-SQL>.

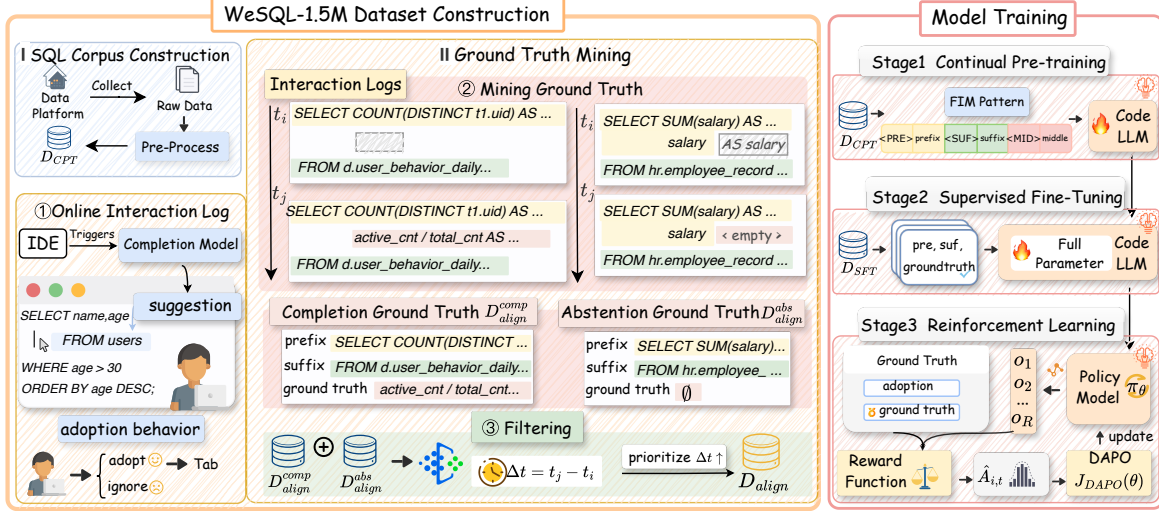


Figure 1: Overview of PACE-SQL.

ated suggestions, revealing genuine user requirements. As shown in Figure 1, by analyzing these logs, we derive two categories of training samples: *completion-required samples* and *abstention samples*. These categories collectively represent user requirements in practical development scenarios.

Mining Online Interaction Logs. Each time the IDE triggers a completion request, the model generates a suggestion $\hat{y}_i$ based on the code context $c_i = (p_i, s_i)$, where p_i and s_i denote the prefix and suffix around the cursor. We record each interaction as $\langle t_i, c_i, \hat{y}_i, a_i \rangle$, where t_i is the timestamp and $a_i = 1$ if user accepts the suggestion via Tab.

Behavioral Patterns of User Interactions. The instantaneous adoption signal a_i alone cannot reliably reflect the user’s true intent, since accepted suggestions may be revised or discarded, while rejected ones may still be manually implemented. By comparing the user’s adoption of the provided suggestion $\hat{y}_i$ at code state c_i with a later state c_j ($t_j > t_i$), we observe four behavioral patterns:

- *Accepted-and-Kept*: the suggestion is adopted and remains unchanged at the target position;
- *Accepted-but-Revised*: the suggestion is initially adopted but later modified or discarded;
- *Rejected-and-Empty*: the suggestion is ignored and no code is inserted at the target position;
- *Rejected-but-Implemented*: the user ignores the suggestion but inserts code at the target position.

Mining Completion-Required Ground Truth. If a new fragment appears at the target position in a later state c_j that was absent in c_i , we infer a gen-

uine completion need exists under the current context. We therefore construct completion-required samples from Accepted-and-Kept, Accepted-but-Revised, and Rejected-but-Implemented patterns, treating the inserted code as the ground-truth g :

$$\langle c, g, \hat{y}, a \rangle \in \mathcal{D}_{\text{align}}^{\text{comp}}. \quad (1)$$

Mining Abstention Ground Truth. Rejected-and-Empty interactions indicate that no completion is needed at the target position. To reduce unnecessary generation, we treat such cases as abstention samples with empty ground truth:

$$\langle c, g = \emptyset, \hat{y}, a \rangle \in \mathcal{D}_{\text{align}}^{\text{abs}}. \quad (2)$$

High-Confidence Filtering. Since code states shortly after t_i may be unstable, for each context c we keep the sample with the largest temporal gap $\Delta t = t_j - t_i$ across $\mathcal{D}_{\text{align}}^{\text{comp}}$ and $\mathcal{D}_{\text{align}}^{\text{abs}}$, yielding the final alignment dataset $\mathcal{D}_{\text{align}}$.

4 Three-Stage Model Training

4.1 Continual Pre-training

Since SQL data is scarce in the pre-training corpora of most code-based LLMs, the understanding of SQL syntax and query logic in these models remains limited. We therefore perform Continual Pre-training (CPT) on the high-quality SQL corpus $\mathcal{D}_{\text{CPT}}$ constructed in Sec. 3.1 under the FIM paradigm. Each SQL statement is randomly partitioned into prefix p , middle m , and suffix s in the PSM format. The training objective is:

$$\mathcal{L}_{\text{CPT}}(\theta) = -\mathbb{E}_{x \sim \mathcal{D}_{\text{CPT}}}[\log P_{\theta}(m \mid p, s)]. \quad (3)$$

4.2 Supervised Fine-Tuning

Although CPT captures SQL syntax regularities, its random FIM sampling misaligns with real completion triggers and ignores abstention. We therefore perform Supervised Fine-Tuning (SFT) on $\mathcal{D}_{\text{SFT}}$, a small subset of $\mathcal{D}_{\text{align}}$ from Sec. 3.2. The model is trained autoregressively to generate the ground-truth g from the context $c = (p, s)$:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\mathbb{E}_{(c,g) \sim \mathcal{D}_{\text{SFT}}} [\log P_{\theta}(g | c)]. \quad (4)$$

4.3 Reinforcement Learning

While CPT and SFT improve SQL completion ability, they do not explicitly optimize for user preferences, such as when to provide suggestions and when to remain silent. Accordingly, we apply reinforcement learning on $\mathcal{D}_{\text{RL}}$, the remaining portion of $\mathcal{D}_{\text{align}}$. Specifically, we design a reward function that incorporates completion correctness, user adoption, and abstention. Optimized with DAPO (Yu et al., 2025), this formulation guides the model to better align with user preferences.

Reward Design. To incorporate user behavioral preferences into reinforcement learning, we propose a unified reward function based on user adoption and model-generated suggestions recorded in interaction logs. Given a code context $c = (p, s)$, the policy samples a completion rollout y . The primary reward component, R_{match} , measures whether the generated completion exactly matches the ground-truth g :

$$R_{\text{match}} = \text{EM}(y, g), \quad (5)$$

where EM (Exact Match) is a binary indicator that returns 1 if $y = g$ and 0 otherwise.

User acceptance ($a = 1$) of the suggestion $\hat{y}$ provides an implicit preference signal. Accordingly, we introduce an auxiliary term, weighted by a coefficient λ , that rewards the EM between the rollout y and the accepted suggestion $\hat{y}$:

$$R_{\text{adopt}} = \lambda a \text{EM}(y, \hat{y}). \quad (6)$$

We further consider the model’s abstention behavior. Correct abstention ($y = g = \emptyset$) is already rewarded by R_{match} . In addition, we introduce a small conservative-abstention bonus for the case ($g \neq \emptyset, y = \emptyset$). Although mismatching the ground truth, abstention is still preferable to a potentially incorrect completion. We therefore assign a small positive reward δ for this behavior:

$$R_{\text{abs}} = \delta \cdot \mathbf{1}[a = 0] \cdot \mathbf{1}[y = \emptyset] \cdot \mathbf{1}[g \neq \emptyset]. \quad (7)$$

The final reward function is defined as:

$$R(y, g, \hat{y}, a) = R_{\text{match}} + R_{\text{adopt}} + R_{\text{abs}}. \quad (8)$$

Unlike conventional rewards based solely on exact match, ours explicitly rewards both useful suggestions and conservative abstention, thereby mitigating over-suggestion and aligning SQL completion with practical user preferences.

DAPO Optimization. We adopt Decoupled Clip and Dynamic Sampling Policy Optimization (DAPO) (Yu et al., 2025) as the policy optimization algorithm. DAPO samples multiple candidate completions and optimizes the policy using group-relative advantages. The objective is defined as:

$$J_{\text{DAPO}}(\theta) = \mathbb{E}_{\mathcal{D}, \{o_i\}_{i=1}^G} \left[\frac{1}{G} \sum_{i=1}^G \sum_{t=1}^{|o_i|} \min \left(r_{i,t}(\theta) \hat{A}_{i,t}, \text{clip}(r_{i,t}(\theta), 1 - \epsilon_{\text{low}}, 1 + \epsilon_{\text{high}}) \hat{A}_{i,t} \right) \right],$$

where $\{o_i\}_{i=1}^G$ are candidate completions from the old policy π_{old} , and the importance ratio is

$$r_{i,t}(\theta) = \frac{\pi_{\theta}(o_{i,t} | q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t} | q, o_{i,<t})}. \quad (9)$$

The advantage $\hat{A}_{i,t}$ is computed by reward normalization within the sampled group:

$$\hat{A}_{i,t} = \frac{R_i - \text{mean}(\{R_i\}_{i=1}^G)}{\text{std}(\{R_i\}_{i=1}^G)}, \quad (10)$$

where R_i is the rollout reward of completion o_i .

5 Experiments

5.1 Experimental Settings

We conduct extensive experiments to evaluate PACE-SQL and answer the following questions: (1) **RQ1:** Does PACE-SQL outperform state-of-the-art SQL completion baselines? (2) **RQ2:** How do CPT, SFT, and RL contribute to the overall performance? (3) **RQ3:** Does the customized reward effectively align the model with user preferences? (4) **RQ4:** What is the impact of abstention data on model performance? (5) **RQ5:** How does PACE-SQL perform in online deployment?

Datasets. We use two datasets, both containing completion-required and abstention samples: (1) **WeSQL-1.5M**, an industrial dataset that we mine from real completion interaction logs, with 1.5M training samples and 4.4K test samples.

Model	WeSQL-1.5M				OpenSQL-FIM			
	EM	ES	ROUGE-L	JW	EM	ES	ROUGE-L	JW
General LLMs								
DS-V4-Flash	29.94 $\pm$ 0.05	50.84 $\pm$ 0.03	48.12 $\pm$ 0.07	68.67 $\pm$ 0.05	60.17 $\pm$ 0.12	78.91 $\pm$ 0.11	77.82 $\pm$ 0.11	88.68 $\pm$ 0.05
Code LLMs								
DS-Coder-6.7B	24.00 $\pm$ 0.13	44.77 $\pm$ 0.10	42.43 $\pm$ 0.12	63.83 $\pm$ 0.08	56.73 $\pm$ 0.12	75.68 $\pm$ 0.09	74.48 $\pm$ 0.11	86.41 $\pm$ 0.07
Qwen-Coder-7B	23.70 $\pm$ 0.11	43.98 $\pm$ 0.08	42.19 $\pm$ 0.09	64.35 $\pm$ 0.06	52.29 $\pm$ 0.10	72.13 $\pm$ 0.08	70.90 $\pm$ 0.09	83.84 $\pm$ 0.06
aiXcoder-7B	21.68 $\pm$ 0.12	39.82 $\pm$ 0.01	37.20 $\pm$ 0.07	56.19 $\pm$ 0.07	48.45 $\pm$ 0.17	69.36 $\pm$ 0.02	67.56 $\pm$ 0.04	81.26 $\pm$ 0.01
StructureCoder-7B	20.11 $\pm$ 0.03	40.15 $\pm$ 0.01	38.44 $\pm$ 0.02	60.57 $\pm$ 0.02	42.93 $\pm$ 0.04	64.48 $\pm$ 0.03	63.97 $\pm$ 0.04	78.90 $\pm$ 0.01
SeedCoder-8B	21.66 $\pm$ 0.02	42.56 $\pm$ 0.11	40.56 $\pm$ 0.11	62.18 $\pm$ 0.10	51.55 $\pm$ 0.10	71.21 $\pm$ 0.03	69.87 $\pm$ 0.07	83.29 $\pm$ 0.01
Qwen-Coder-14B	27.52 $\pm$ 0.09	47.10 $\pm$ 0.08	45.22 $\pm$ 0.09	65.86 $\pm$ 0.06	56.69 $\pm$ 0.10	75.46 $\pm$ 0.08	74.07 $\pm$ 0.09	85.85 $\pm$ 0.06
StarCoder2-15B	19.64 $\pm$ 0.11	39.34 $\pm$ 0.02	37.33 $\pm$ 0.04	59.16 $\pm$ 0.02	45.18 $\pm$ 0.04	66.38 $\pm$ 0.05	65.91 $\pm$ 0.04	79.32 $\pm$ 0.03
Qwen-Coder-32B	28.64 $\pm$ 0.09	48.64 $\pm$ 0.08	46.96 $\pm$ 0.08	67.26 $\pm$ 0.05	58.74 $\pm$ 0.09	77.28 $\pm$ 0.07	76.15 $\pm$ 0.08	87.17 $\pm$ 0.06
DS-Coder-33B	24.44 $\pm$ 0.09	45.52 $\pm$ 0.07	43.55 $\pm$ 0.08	64.52 $\pm$ 0.05	58.79 $\pm$ 0.09	77.46 $\pm$ 0.07	76.33 $\pm$ 0.08	87.62 $\pm$ 0.05
External Filter								
FrugalCoder (DS-6.7B)	22.91 $\pm$ 0.12	38.14 $\pm$ 0.09	36.31 $\pm$ 0.13	52.03 $\pm$ 0.07	54.75 $\pm$ 0.11	71.59 $\pm$ 0.10	70.29 $\pm$ 0.10	80.80 $\pm$ 0.08
FrugalCoder (Qwen-14B)	25.61 $\pm$ 0.09	40.52 $\pm$ 0.07	39.05 $\pm$ 0.10	54.59 $\pm$ 0.05	54.97 $\pm$ 0.11	71.75 $\pm$ 0.07	70.21 $\pm$ 0.08	80.55 $\pm$ 0.07
Our Models								
PACE-SQL (DS-6.7B)	36.54 $\pm$ 0.12	56.66 $\pm$ 0.05	56.06 $\pm$ 0.07	70.29 $\pm$ 0.06	61.91 $\pm$ 0.05	78.91 $\pm$ 0.03	78.26 $\pm$ 0.02	90.01 $\pm$ 0.01
PACE-SQL (Qwen-14B)	39.61 $\pm$ 0.08	58.36 $\pm$ 0.06	56.44 $\pm$ 0.07	76.13 $\pm$ 0.04	63.87 $\pm$ 0.09	79.28 $\pm$ 0.07	79.17 $\pm$ 0.08	90.06 $\pm$ 0.05

Table 1: Performance comparison on WeSQL-1.5M and OpenSQL-FIM. Entries with “ $\pm$ ” are reported as mean $\pm$ std over three independent runs. **Best** and second-best per column are highlighted.

Note. Abbreviations: DS = DeepSeek; Qwen-Coder = Qwen2.5-Coder.

(2) **OpenSQL-FIM**, derived from The Stack V2 (Lozhkov et al., 2024) and processed into FIM instances, containing 1.15M training samples and 4.2K test samples. Construction details are provided in Appendix H.

Baselines. We compare PACE-SQL with three baseline categories: (1) **General LLMs**: We select DeepSeek-V4-Flash, the large-scale LLMs that support FIM. (2) **Code LLMs**: We evaluate various parameter scales across DeepSeek-Coder (Guo et al., 2024), Qwen-Coder (Hui et al., 2024), SeedCoder (Seed et al., 2025), StarCoder2 (Lozhkov et al., 2024), aiXcoder (Jiang et al., 2025), and StructureCoder (Ren et al., 2025). (3) **External Filter**: FrugalCoder (Sun et al., 2024), which trains an estimator to suppress low-confidence completions. Details of the baselines and reproduction are provided in Appendix K.

Training Configurations. We use Qwen2.5-Coder-14B and DeepSeek-Coder-6.7B as backbones and train two variants with distinct data sources: (1) an *industrial* variant trained and evaluated on WeSQL-1.5M; and (2) an *open-source* variant trained and evaluated on OpenSQL-FIM, using only R_{match} during RL as open-source corpora lack user adoption signals and interaction information. Hyperparameters and hardware configurations are detailed in Appendix I.

Evaluation Metrics. We employ four metrics (all higher is better) to assess completion quality: Exact Match (EM), Edit Similarity (ES), ROUGE-L, and Jaro–Winkler (JW) similarity (Jaro, 1989). Detailed definitions are provided in Appendix B.

5.2 Main Results (RQ1)

Table 1 reports performance comparisons on two benchmarks. We summarize three observations:

First, PACE-SQL consistently outperforms all baselines. PACE-SQL (Qwen-14B) surpasses DeepSeek-V4-Flash by 9.67% on WeSQL-1.5M and 3.70% on OpenSQL-FIM in EM. Notably, PACE-SQL yields consistent gains on two base models across sizes, indicating that its benefits do not depend on a particular backbone. Second, PACE-SQL achieves larger gains on interaction logs than on static open-source data. This is expected since interaction logs better reflect user needs, while the consistent gains on OpenSQL-FIM show that our framework remains effective on public SQL data. Third, the external filter yields a slight degradation relative to its backbone: FrugalCoder only suppresses low-confidence outputs without improving completion quality. In contrast, PACE-SQL internalizes abstention into training and jointly improves completion quality. Analyses of completion and abstention cases are provided in Appendix D and Appendix E, respectively.

Model Setting	EM	ES	ROUGE-L	JW
<i>WeSQL-1.5M</i>				
Qwen2.5-Coder	27.52	47.10	45.22	65.86
+ CPT	30.23	51.69	49.37	70.02
+ SFT	36.57	53.82	56.19	75.62
+ RL (Ours)	39.61	58.36	56.44	76.13
<i>OpenSQL-FIM</i>				
Qwen2.5-Coder	56.69	75.46	74.07	85.85
+ CPT	58.67	77.41	76.13	87.43
+ SFT	62.13	80.68	79.55	90.35
+ RL (Ours)	63.87	79.28	79.17	90.06

Table 2: Ablation study of training stages on Qwen2.5-Coder. Best results are in bold.

5.3 Additional Experiments

We further analyze the contributions of each component and online performance of PACE-SQL using Qwen2.5-Coder-14B as the backbone model.

Impact of Training Stages (RQ2). To isolate the contribution of each training stage, we conduct a progressive ablation study, with results reported in Table 2. CPT and SFT consistently improve ES, JW and ROUGE-L on both datasets, showing their effectiveness in SQL-domain adaptation. RL further yields EM gains. On OpenSQL-FIM, this EM improvement comes with a slight drop in similarity-based metrics. We attribute this drop to the abstention behavior learned during RL. In uncertain completion-required cases, SFT tends to produce a partially matching but EM-incorrect completion, which still earns similarity credit through character overlap. RL is more likely to abstain, losing this similarity credit without changing EM on these examples, which reduces unhelpful interruptions to users in reality.

Effectiveness of Reward (RQ3). To evaluate the effectiveness of our proposed reward function, we conduct an ablation study on WeSQL-1.5M. We compare three variants: (a) R_{match} , (b) $R_{match} + R_{adopt}$, and (c) the full reward. As shown in Figure 2, removing any component degrades at least one metric, while the full reward achieves the best results on all metrics. This confirms that match correctness, user adoption, and conservative abstention provide complementary signals for alignment.

Impact of Abstention Data (RQ4). We study the effect of abstention data mined from user interaction logs on WeSQL-1.5M. To decouple data supervision from reward engineering, we only apply R_{match} in two settings: (a) w/o abstention data

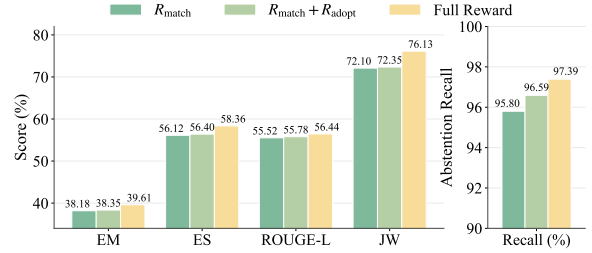


Figure 2: Ablation study for the reward components. Abstention Recall is the probability of producing an empty prediction when the ground truth is empty.

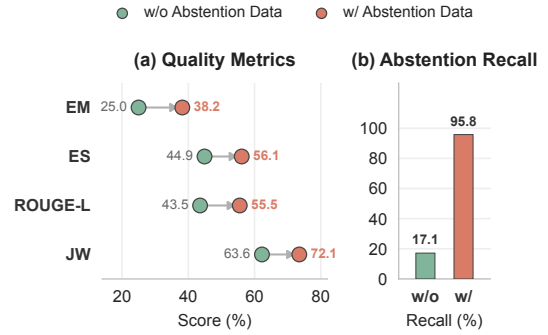


Figure 3: Ablation study for abstention data, comparing data with and without 20% abstention samples.

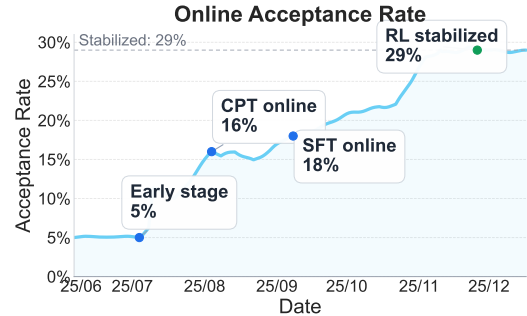


Figure 4: User acceptance rate of PACE-SQL across different deployment stages.

and (b) with 20% abstention data. Figure 3 shows that abstention data improves all metrics and raises Abstention Recall from 17.1% to 95.8%, indicating that such samples provide critical supervision for recognizing no-completion contexts even without the full reward function.

Online Deployment (RQ5). Figure 4 presents the online user acceptance rate during PACE-SQL deployment. Starting from a vanilla Qwen2.5-Coder-7B baseline with only 5% acceptance, the incremental introduction of CPT, SFT, and RL progressively raises the acceptance rate to 29%. Deployment details are provided in Appendix J.

To further investigate the factors underlying the acceptance-rate improvement across deployment

Stage	Req.	Disp.	Acc.	Acc. Tok.
Early Stage	44.62	17.98	0.84	7.53
CPT	158.59	16.38	2.61	20.19
SFT	155.12	15.47	2.88	25.35
RL	369.50	18.35	5.32	37.67

Table 3: Online deployment statistics averaged per user-day across stages. Req. denotes completion requests; Disp., displayed suggestions; Acc., suggestions accepted via Tab; and Acc. Tok., tokens contained in Tab-accepted suggestions.

stages, we report the average per-user-day statistics in Table 3. As the model’s abstention ability improves, empty outputs reduce inference costs and allow the system to handle more IDE requests, while the number of suggestions displayed to each user per day remains stable. Meanwhile, the numbers of Tab-accepted suggestions and accepted tokens per user-day increase substantially. These results show that PACE-SQL improves completion quality, raising the acceptance rate by increasing the number of accepted suggestions rather than reducing the number of displayed suggestions.

6 Conclusion

We propose PACE-SQL, a preference-aligned framework for SQL completion that addresses over-suggestion, random FIM sampling, and scarce SQL datasets. We build two SQL completion datasets, WeSQL-1.5M and OpenSQL-FIM. We adopt a CPT-SFT-RL training pipeline with a unified reward to jointly optimize correctness, adoption, and abstention. PACE-SQL has been deployed in our enterprise IDE for SQL completion.

Limitations

While PACE-SQL demonstrates significant improvements in SQL code completion, we acknowledge certain scope boundaries that present opportunities for future research. First, our current framework primarily focuses on single-line completions within the active file. Although this covers most real-world IDE interactions, complex SQL queries often rely on table schemas scattered across other files in the repository. Incorporating repository-level context could therefore further enhance completion accuracy for multi-table queries. Second, to avoid interrupting real development workflows, we obtain abstention signals by comparing two code states within the time window rather than asking developers to explicitly la-

bel each case through pop-up prompts. Although this Δt -based construction yields high-confidence labels, such behavioral proxies cannot completely represent true user intent.

Ethical Considerations

The training data of PACE-SQL consists of SQL scripts and completion interaction logs collected from an internal enterprise data platform. The platform is accessible only to internal employees, with no external users. Before entering the training pipeline, the data is desensitized to remove personally identifiable information such as user names, account identifiers, and contact details embedded in SQL. We further note that our logging is strictly bounded to what the completion service already requires: the system does not observe the user’s IDE interface, and the behavioral patterns in Sec. 3.2 are reconstructed solely from the cursor-side code context $c_i = (p_i, s_i)$ transmitted at each completion request, together with the resulting suggestion and adoption flag.

Acknowledgments

This work was sponsored by National Key Research and Development Program of China (2023YFB2703604) and National Natural Science Foundation of China (62472327). We thank Tencent TEG Data Computing Platform Department for its valuable support throughout this work.

References

- Gal Bakal, Ali Dasdan, Yaniv Katz, Michael Kaufman, and Guy Levin. 2025. [Experience with GitHub Copilot for Developer Productivity at Zoominfo](#). *Preprint*, arXiv:2501.13282.
- Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. [Efficient Training of Language Models to Fill in the Middle](#). *Preprint*, arXiv:2207.14255.
- A.Z. Broder. 1997. [On the resemblance and containment of documents](#). In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others.

2021. [Evaluating Large Language Models Trained on Code](#). *Preprint*, arXiv:2107.03374.
- Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sashank Tszyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, and 48 others. 2023. PaLM: scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24(1).
- Aral de Moor, Yana Hrynevich, Hleb Badzeika, Vladyslav Furda, Marko Kojic, Artem Savelev, Kostadin Cvejovski, Darya Rovdo, and Ekaterina Garanina. 2026. [Control Models for In-IDE Code Completion](#). In *Proceedings of the 3rd ACM/IEEE International Workshop on Integrated Development Environments*, page 3439. ACM.
- Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CROSS-CODEEVAL: a diverse and multilingual benchmark for cross-file code completion. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. [DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence](#). *Preprint*, arXiv:2401.14196.
- Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. [Large Language Models for Software Engineering: A Systematic Literature Review](#). *ACM Trans. Softw. Eng. Methodol.*, 33(8).
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. [Qwen2.5-Coder Technical Report](#). *Preprint*, arXiv:2409.12186.
- Piotr Indyk and Rajeev Motwani. 1998. [Approximate nearest neighbors: towards removing the curse of dimensionality](#). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 604613, New York, NY, USA. Association for Computing Machinery.
- Matthew A Jaro. 1989. Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. *Journal of the American Statistical association*, 84(406):414–420.
- Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. [A Survey on Large Language Models for Code Generation](#). *ACM Trans. Softw. Eng. Methodol.*, 35(2).
- Siyuan Jiang, Jia Li, He Zong, Huanyu Liu, Hao Zhu, Shukai Hu, Erlu Li, Jiazheng Ding, Yu Han, Wei Ning, and Ge Li. 2025. [aiXcoder-7B: A Lightweight and Effective Large Language Model for Code Processing](#). In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 215–226.
- Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2022. [Deduplicating training data makes language models better](#). In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8424–8445, Dublin, Ireland. Association for Computational Linguistics.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. [StarCoder: may the source be with you!](#) *Preprint*, arXiv:2305.06161.
- Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. [StarCoder 2 and The Stack v2: The Next Generation](#). *Preprint*, arXiv:2402.19173.
- Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. [When to show a suggestion? integrating human feedback in AI-assisted programming](#). In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'24/IAAI'24/EAAI'24. AAAI Press.
- Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. [CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis](#). *Preprint*, arXiv:2203.13474.
- OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. [GPT-4 Technical Report](#). *Preprint*, arXiv:2303.08774.

- James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. [“It’s Weird That it Knows What I Want”: Usability and Interactions with Copilot for Novice Programmers.](#) *ACM Trans. Comput.-Hum. Interact.*, 31(1).
- Houxing Ren, Zimu Lu, Weikang Shi, Haotian Hou, Yunqiao Yang, Ke Wang, Aojun Zhou, Juntong Pan, Mingjie Zhan, and Hongsheng Li. 2025. [Alignment with fill-in-the-middle for enhancing code generation.](#) In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 8304–8320, Suzhou, China. Association for Computational Linguistics.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. [Code Llama: Open Foundation Models for Code.](#) *Preprint*, arXiv:2308.12950.
- ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, and 8 others. 2025. [Seed-Coder: Let the Code Model Curate Data for Itself.](#) *Preprint*, arXiv:2506.03524.
- Liang Shi, Zhengju Tang, Nan Zhang, Xiaotong Zhang, and Zhi Yang. 2025. [A Survey on Employing Large Language Models for Text-to-SQL Tasks.](#) *ACM Comput. Surv.*, 58(2).
- Zhensu Sun, Xiaoning Du, Fu Song, Shangwen Wang, Mingze Ni, Li Li, and David Lo. 2024. [Don’t Complete It! Preventing Unhelpful Code Completion for Productive and Sustainable Neural Code Completion Systems.](#) *ACM Trans. Softw. Eng. Methodol.*, 34(1).
- Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. [Practitioners’ Expectations on Code Completion.](#) *Preprint*, arXiv:2301.03846.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. [Qwen3 Technical Report.](#) *Preprint*, arXiv:2505.09388.
- Qiyang Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, and 16 others. 2025. [DAPO: An Open-Source LLM Reinforcement Learning System at Scale.](#) *Preprint*, arXiv:2503.14476.

A Related Work

Code Completion with LLMs. Early models such as Codex, CodeGen, and PaLM (Chen et al., 2021; Nijkamp et al., 2023; Chowdhery et al., 2023), followed a left-to-right next-token prediction paradigm, which is less suitable for infilling scenarios that require both left and right context. The Fill-in-the-Middle (FIM) (Bavarian et al., 2022) addresses this by training models to generate missing code conditioned on surrounding context, and has been adopted by modern code LLMs such as CodeLlama, DeepSeek-Coder, and Qwen-Coder (Li et al., 2023; Rozière et al., 2024; Guo et al., 2024; Hui et al., 2024; Yang et al., 2025).

Addressing Unhelpful Suggestions in Code Completion. Existing methods mostly rely on external decision models to mitigate unhelpful suggestions. FrugalCoder (Sun et al., 2024) trains an estimator to suppress low-confidence completion context. Microsoft and JetBrains (Mozannar et al., 2024; de Moor et al., 2026) adopt a two-stage strategy in which a trigger model decides whether to invoke the LLM and a filter model decides whether to display the generated suggestion, which inevitably increases the complexity of the overall framework. In contrast, rather than filtering out bad suggestions, PACE-SQL avoids generating them at the source by modeling abstention as an internal generation behavior.

B Evaluation Metrics Details

For evaluation parameters, we set the maximum sequence length to 16K tokens and limit the maximum generated length to 60 tokens. We employ four metrics (all higher is better) to assess completion quality:

- (1) **Exact Match (EM):** Measures character-level identity with the ground truth.
- (2) **Edit Similarity (ES):** Quantifies code revision cost via normalized Levenshtein distance.
- (3) **ROUGE-L:** Captures sequence-level overlap via the longest common subsequence.
- (4) **Jaro-Winkler (JW):** A prefix-weighted string similarity metric, computed as:

$$JW(y, g) = J(y, g) + \ell \cdot p \cdot (1 - J(y, g)), \quad (11)$$

where J is the Jaro similarity, ℓ the common prefix length (up to 4), and $p = 0.1$.

Model	EM	ES
<i>Closed-Source LLMs</i>		
GPT-4o	20.22	28.75
Claude-Sonnet-4-6	21.00	34.05
<i>Open-Source Code LLMs (Base)</i>		
Qwen2.5-Coder-14B	56.69	75.46
DeepSeek-Coder-6.7B	56.73	75.68

Table 4: Performance of closed-source LLMs on OpenSQL-FIM. Two open-source code LLMs are included for reference.

C Closed-Source LLMs on SQL FIM

In code completion scenarios, practitioners typically prefer base models over instruction-tuned models for FIM tasks. This is because alignment training during the instruction tuning stage significantly alters the model’s output distribution, biasing it toward natural-language-style responses rather than precise code completions, thereby degrading completion accuracy. Accordingly, we prefer base models for open-source baselines in our main experiments.

Current mainstream closed-source LLMs (e.g., GPT-4o, Claude-Sonnet-4-6) only provide instruction-based API interfaces, which neither support FIM-specific special tokens nor provide access to base model variants. Therefore, we design a natural-language prompt template (illustrated in Figure 6) that embeds the prefix and suffix in a structured format to evaluate these models on FIM tasks.

Due to privacy constraints on industrial data, we evaluate closed-source models exclusively on the publicly constructed dataset, OpenSQL-FIM. As shown in Table 4, both GPT-4o and Claude-Sonnet-4-6 fall substantially behind even 6.7B-scale open-source code LLMs on EM, confirming that prompt-based FIM simulation on instruction-tuned models cannot substitute for native FIM capability in code completion.

D Completion-Required Analysis

We separately examine how PACE-SQL performs when a completion is genuinely required. The WeSQL-1.5M test set contains 4,400 cases, of which 80% (3,519) are completion-required and the remaining 20% are abstention cases.

Model	EM	Gen.	Correct
Qwen2.5-Coder-32B	0.2043	3251	719
DeepSeek-V4-Flash	0.2160	3300	760
PACE-SQL (Qwen-14B)	0.2515	2706	885

Table 5: Performance on the completion-required subset (3,519 cases) of the WeSQL-1.5M test set. Gen. denotes generated completions, and Correct denotes correct completions.

Pattern	Cases	Correct
Accepted-and-Kept	119 (3.38%)	80
Accepted-but-Revised	178 (5.06%)	45
Rejected-but-Implemented	3,222 (91.56%)	760

Table 6: Distribution of the three behavioral patterns among the completion-required cases in the WeSQL-1.5M test set, with the number of correct completions produced by PACE-SQL (Qwen-14B).

Quality on Completion-Required Cases. On the completion-required subset, we compare PACE-SQL with the best general LLM baseline (DeepSeek-v4-flash) and the strongest code LLM baseline (Qwen2.5-Coder-32B). As shown in Table 5, although PACE-SQL generates fewer completions, it produces more correct completions and achieves higher EM than both baselines. This is valuable in practice, as it indicates that PACE-SQL tends to present useful suggestions to users while reducing unnecessary interruptions.

Gains Beyond the Predecessor Model. As described in Sec. 3.2, completion-required cases are mined from three patterns: Accepted-and-Kept, Accepted-but-Revised, and Rejected-but-Implemented. To verify that PACE-SQL genuinely improves completion ability rather than being a clone of the predecessor model’s well-served subset, we decompose the completion-required cases by these three patterns, with results reported in Table 6. Accepted-and-Kept cases, which directly reflect the predecessor’s adopted suggestions, form only a small fraction, whereas Rejected-but-Implemented cases dominate and PACE-SQL correctly completes 760 of them. This confirms that PACE-SQL’s improvements go well beyond replicating the predecessor model’s outputs.

Comparison with an Ideal Classifier. To examine whether external gating can replace internalized abstention, we construct an upper

Model	EM	Gen.	Correct Comp.	Correct Abst.
Ideal Classifier	0.3593	3262	700	881
PACE-SQL	0.3961	2706	885	858

Table 7: Comparison with the ideal upper bound of external gating on the WeSQL-1.5M test set, using Qwen2.5-Coder-14B as the backbone. Gen. denotes generated completions on completion-required cases; Correct Comp. and Correct Abst. denote correct completions and correct abstentions, respectively.

bound using an ideal classifier paired with a frozen Qwen2.5-Coder-14B backbone. The classifier always displays suggestions for completion-required cases and abstains on all abstention cases. As shown in Table 7, even with a perfect classifier, the frozen backbone underperforms PACE-SQL. This demonstrates that PACE-SQL not only learns when to provide suggestions but also improves completion quality.

E Abstention Recall Across Models

A core motivation of PACE-SQL is to suppress unhelpful completions on prompts whose ground-truth target is empty. Complementary to the completion-required analysis above, we report *Abstention Recall* on both datasets, defined as the proportion of empty-target prompts on which a model also produces an empty completion, i.e., $P(y = \emptyset \mid g = \emptyset)$. A higher value indicates a stronger ability to remain silent when no useful completion is expected.

Table 8 reports the Abstention Recall of all baselines and PACE-SQL on WeSQL-1.5M and OpenSQL-FIM. We observe three patterns. First, existing backbone models show limited abstention ability on WeSQL-1.5M, where all general and code LLM baselines remain below 65% Abstention Recall. On OpenSQL-FIM, abstention recall is generally higher, with DeepSeek-V4-Flash reaching 81.79%, while most code LLMs remain below 80%. Second, FrugalCoder improves Abstention Recall over its corresponding backbones, suggesting that an explicit pre-filtering estimator can help suppress completions on empty-target prompts. Third, PACE-SQL obtains higher Abstention Recall than the compared baselines on both benchmarks. This suggests that incorporating abstention behavior into training can improve the model’s tendency to remain silent when no completion is expected.

Model	WeSQL-1.5M	OpenSQL-FIM
DS-V4-Flash	64.70	81.79
Qwen2.5-Coder-32B	61.41	78.26
Qwen2.5-Coder-14B	58.00	75.06
DS-Coder-33B	50.23	78.15
StarCoder2-15B	35.64	48.79
aiXcoder-7B	53.30	69.43
SeedCoder-8B	41.20	64.24
Qwen2.5-Coder-7B	49.60	66.34
DS-Coder-6.7B	49.77	75.50
FrugalCoder (Qwen-14B)	67.88	77.59
FrugalCoder (DS-6.7B)	61.82	77.81
PACE-SQL (DS-6.7B)	88.75	97.02
PACE-SQL (Qwen-14B)	97.39	98.45

Table 8: Abstention Recall (%) on WeSQL-1.5M and OpenSQL-FIM. Higher is better.

Model	EM	ES	ROUGE-L	JW
Qwen2.5-Coder	56.69	75.46	74.07	85.85
Qwen2.5-Coder (+50K OpenSQL-FIM)	56.71	73.40	74.01	86.46
PACE-SQL (+50K OpenSQL-FIM)	<u>62.63</u>	<u>78.32</u>	<u>78.31</u>	<u>89.35</u>
PACE-SQL (OpenSQL-FIM Only)	63.87	79.28	79.17	90.06

Table 9: Cross-dataset transferability on OpenSQL-FIM. PACE-SQL (*OpenSQL-FIM Only*) is trained on the full OpenSQL-FIM training data. PACE-SQL (+50K *OpenSQL-FIM*) continues RL from the PACE-SQL model trained on WeSQL-1.5M, using only 50K additional OpenSQL-FIM samples ($\sim 20\%$ of the training set) with R_{match} .

F Cross-Dataset Generalization

We further assess the cross-dataset transferability of PACE-SQL by evaluating it on OpenSQL-FIM. As shown in Table 9, starting from the PACE-SQL model trained solely on our enterprise dataset WeSQL-1.5M, we continue RL training on only 50K OpenSQL-FIM samples ($\sim 20\%$ of the 250K RL training set). Additionally, we apply the same RL training to Qwen2.5-Coder-14B using the same 50K samples. The resulting PACE-SQL model reaches an EM of 62.63%, substantially outperforming the Qwen2.5-Coder backbone and approaching the performance of the model trained on the full OpenSQL-FIM dataset. This indicates that, by building on the SQL completion and abstention ability acquired from WeSQL-1.5M training, PACE-SQL can efficiently adapt to a new domain and nearly match full-data performance with only a small amount of target-domain supervision.

G WeSQL-1.5M Construction Details

We collect raw SQL submission scripts from our enterprise data warehouse and governance platform, where SQL is the dominant language among over 8.7K users and the system serves over 30K SQL completion requests per day. However, raw SQL data is inherently noisy, containing syntax errors, debugging fragments, and redundant query templates. To address this, we apply a multi-stage cleaning pipeline consisting of *length filtering*, *deduplication*, and *syntax validation*.

Length Filtering. Let $\mathcal{D}_{\text{raw}}$ denote the initial collection of SQL scripts. We retain only scripts with character length in $[30, 50,000]$, filtering out incomplete fragments and excessively long submissions, yielding the filtered subset $\mathcal{D}_1$.

Deduplication. The filtered corpus $\mathcal{D}_1$ still contains many near-duplicate scripts that differ only in minor lexical variations. For example, two scripts `SELECT name FROM employees WHERE dept = 'sales'` and `SELECT name FROM employees WHERE dept = 'marketing'` share identical structure and differ only in a single literal value. Retaining these highly similar samples may cause overfitting to frequent query templates.

Following prior work (Lee et al., 2022), we use MinHash (Broder, 1997) with Locality-Sensitive Hashing (LSH) (Indyk and Motwani, 1998) for approximate deduplication, yielding a deduplicated corpus $\mathcal{D}_2$. The core idea is to hash each SQL into a compact signature to approximate similarity, then partition signatures into buckets to efficiently identify near-duplicate pairs. Near-duplicate scripts are grouped into clusters, and a single representative is retained from each cluster, yielding $\mathcal{D}_2 \subseteq \mathcal{D}_1$.

Syntactic Validation. To ensure syntactic correctness, we further parse each script in $\mathcal{D}_2$ using `sqlglot` and discard parsing failures, obtaining the final continual pre-training corpus $\mathcal{D}_{\text{WeSQL-CPT}}$.

Table 10 summarizes the data composition of WeSQL-1.5M. The continual pre-training corpus $\mathcal{D}_{\text{WeSQL-CPT}}$ is collected in the stage described in Sec. 3.1, while the supervised fine-tuning set $\mathcal{D}_{\text{WeSQL-SFT}}$ and the reinforcement learning set $\mathcal{D}_{\text{WeSQL-RL}}$ are both obtained in the stage described in Sec. 3.2. The held-out test data contains 4,400 real user FIM interactions and is used exclusively for evaluation.

Stage / Subset	# Samples
<i>Training Allocation</i>	
Continual Pre-training ($\mathcal{D}_{\text{WeSQL-CPT}}$)	900,000
Supervised Fine-tuning ($\mathcal{D}_{\text{WeSQL-SFT}}$)	300,000
Reinforcement Learning ($\mathcal{D}_{\text{WeSQL-RL}}$)	300,000
Held-out Test	4,400

Table 10: Statistics of the WeSQL-1.5M. The upper section shows the data allocation across training stages; the held-out test data is used exclusively for evaluation.

H OpenSQL-FIM Construction Details

Raw SQL Preprocessing. We preprocess the raw SQL corpus from StackV2 (Lozhkov et al., 2024) through three steps:

- *Length Filtering*: we retain only SQL statements whose character length falls between 30 and 50,000.
- *Deduplication*: we apply MinHash-LSH to remove near-duplicate scripts, following the same procedure described in Appendix G.
- *Syntactic Validation*: we parse each SQL statement and retain only those that pass compilation.

The data statistics at each stage are shown in Table 11. We globally shuffle the cleaned corpus and first hold out a test set of 4,188 samples, of which 20% are abstention test cases. The remaining data is then randomly split into CPT, SFT, and RL subsets, ensuring **no overlap** between the test set and any training partition.

FIM Sample Construction. For each SQL statement, we randomly split it into a prefix p , a middle span m , and a suffix s , where m serves as the ground-truth target. We then serialize each FIM sample using the model-specific FIM format of the corresponding backbone.

For Qwen2.5-Coder, each training sample follows its native FIM tokenization format:

```
<|fim_prefix|> p <|fim_suffix|> s
<|fim_middle|> m.
```

For DeepSeek-Coder-6.7B, each training sample follows its native FIM format:

```
<|fim_begin|> p <|fim_hole|> s
<|fim_end|> m.
```

In real-world IDEs, code completion is only triggered when the cursor lands at a token boundary, where the subsequent character is whitespace

Stage / Subset	# Samples
<i>Corpus Filtering</i>	
Initial collection	5,292,995
→ Deduplication	3,215,473
→ Syntactic Validation	1,245,559
<i>Training Allocation</i>	
Continual Pre-training ($\mathcal{D}_{\text{OpenSQL-FIM-CPT}}$)	600,000
Supervised Fine-tuning ($\mathcal{D}_{\text{OpenSQL-FIM-SFT}}$)	300,000
Reinforcement Learning ($\mathcal{D}_{\text{OpenSQL-FIM-RL}}$)	250,000
Held-out Test	4,188

Table 11: Statistics of the OpenSQL-FIM construction pipeline. The upper section shows the corpus size after each filtering stage; the lower section shows the data allocation across training stages.

or punctuation, rather than in the middle of an identifier. To align the training distribution with this behavior, we require the first character of s to be whitespace or punctuation; splits where it is alphanumeric are resampled up to 100 times, and the sample is discarded if no valid split is found.

Empty Ground-Truth Construction. We explicitly construct empty ground-truth samples to simulate abstention scenarios. A portion of the SFT and RL samples use an empty string as the ground truth, enabling the model to learn when to generate and when to abstain. The proportion of empty ground-truth samples is 20%. Figure 5 shows two representative FIM samples.

Benchmark Confidence Filtering. For the benchmark split, we apply confidence filtering to remove low-quality FIM samples. The filtering consists of two steps:

- *Noise filtering*: we remove samples that are unlikely to represent valid SQL completion scenarios, including samples dominated by non-SQL content, samples with little SQL context around the cursor, and samples containing long non-Chinese/English comments.
- *Preference-consistency filtering*: we use an LLM-as-a-judge to verify whether each sample is consistent with known user behaviors in SQL code completion. Specifically, we sample 3,000 real user interactions from WeSQL-1.5M and estimate the cursor-level completion and abstention rates from them, summarized in Table 12. These rates serve as the judging criteria for filtering OpenSQL-FIM samples. Following the prompt in Figure 7, the judge assigns each sample to *accept*, *borderline*, or *reject*, and we retain the former two.

Cursor State	Description	Non-completion Rate	User Preference
<i>after comma</i>	Prefix ends with a comma	1.4%	Strong completion
<i>right-is-closer</i>	Suffix starts with a closing symbol, e.g., <code>)</code> , <code>']</code> , or <code>"</code>	2.7%	Strong completion
<i>blank line</i>	Cursor is on an empty or indentation-only line	3.7%	Strong completion
<i>other</i>	Other regular cursor contexts	8.8%	Neutral
<i>at EOF</i>	Suffix is empty or nearly empty	12.7%	Weak completion
<i>glued between tokens</i>	Cursor is between two non-whitespace tokens	23.1%	Weak abstention
<i>after semicolon</i>	Prefix ends with a completed statement	25.0%	Strong abstention

Table 12: User completion and abstention preferences summarized by an LLM from WeSQL-1.5M real-user interactions. Non-completion rate denotes the proportion of abstained or rejected completions under each cursor state.

Completion Required Sample	
prefix:	<code>char(50), IN address char(50), IN phone char(12)) BEGIN IF NOT exists (Select * From tbl_borrower b where b.name = name and b.address = address and b.phone = phone) Then INSERT INTO tbl_borrower(nam</code>
suffix:	<code>VALUES(name, address, phone); END IF; END</code>
ground truth:	e, address, phone)

Abstention Sample	
prefix:	<code>CREATE TABLE ru_followers (username VARCHAR, follows VARCHAR)</code>
suffix:	<code>INSERT INTO ru_followers VALUES ('qwer', 'keth')</code>
ground truth:	< empty >

Figure 5: Two representative FIM samples constructed from the OpenSQL-FIM corpus.

I Training Details

All training stages are conducted on $8 \times$ NVIDIA H20 GPUs. We instantiate PACE-SQL with two backbone models, Qwen2.5-Coder-14B and DeepSeek-Coder-6.7B, and apply the same three-stage training pipeline to each backbone. For each backbone, the stages are executed sequentially: CPT starts from the base model, while SFT and RL initialize from the checkpoints produced by their preceding stages, thereby preserving and progressively refining the acquired capabilities. Tables 13 and 14 summarize the training hyperparameters for Qwen2.5-Coder-14B and DeepSeek-Coder-6.7B.

J Online Deployment Details

Setup. We deploy PACE-SQL in three stages on the internal data platform, replacing the prior production model with our CPT, SFT, and RL checkpoints in turn. The platform is restricted to internal employees, and its active user population remained stable throughout the study. We therefore report the online numbers as a pre-/post-deployment comparison on the same cohort, where the trigger policy and suggestion-display UI are kept fixed across stages so that only the underlying model changes.

Result. The platform serves over 8.7K SQL users and over 30K completion requests per day, which provides sufficient daily samples for a stable estimate of the user acceptance rate. A suggestion is considered accepted if the user presses the Tab key, and the acceptance rate is computed as (Tab-accepted / displayed suggestions). Under this setting, as shown in Figure 4, the acceptance rate rises monotonically from 5% on the prior production model to 16% after CPT and 18% after SFT, and finally reaches **29%** with the complete PACE-SQL pipeline (CPT+SFT+RL). This trend is consistent with the incremental contribution of each training stage in real deployment.

K Baseline Details and Reproduction

K.1 aiXcoder-7B

aiXcoder-7B (Jiang et al., 2025) is a 7B decoder-only model trained on 1.2T tokens with three alternating training objectives. Its key novelty is a structured variant of fill-in-the-middle (FIM) prediction. Standard FIM masks a randomly chosen contiguous token span. aiXcoder-7B instead parses each source file into a syntax tree with tree-sitter and randomly picks a non-root, non-leaf node. It then selects a span inside the corresponding code region, forcing the span’s end

Stage	Hyperparameter	Industrial	Open-Source
CPT	Training samples	900K	600K
	Learning rate		2e-5
	Epochs		2
	Max seq length		16,384
	Optimizer		AdamW
	Training time		~72–96 hours
SFT	Training samples	300K	300K
	Learning rate		2e-5
	Epochs		2
	Max seq length		16,384
	Optimizer		AdamW
	Training time		~24–48 hours
RL	Training prompts	300K	250K
	Rollouts per prompt		10
	Temperature / top- p		1.05 / 0.95
	Max prompt length		15,360
	Max response length		128
	Adoption weight λ	0.5	—
	Abstention bonus δ	0.1	—
	Reward function	Full Reward	EM
	Training time		~48–72 hours

Table 13: Training hyperparameters for PACE-SQL with Qwen2.5-Coder-14B as the backbone. “—” marks inapplicable settings: the open-source variant uses only R_{match} during RL, as open-source corpora lack user adoption signals, and thus has neither adoption weight λ nor abstention bonus δ .

Stage	Hyperparameter	Industrial	Open-Source
CPT	Training samples	900K	600K
	Learning rate		2e-5
	Epochs		2
	Max seq length		16,384
	Optimizer		AdamW
	Training time		29 hours
SFT	Training samples	300K	300K
	Learning rate		2e-5
	Epochs		2
	Max seq length		16,384
	Optimizer		AdamW
	Training time		27 hours
RL	Training prompts	300K	250K
	Rollouts per prompt		10
	Temperature / top- p		1.05 / 0.95
	Max prompt length		15,360
	Max response length		128
	Adoption weight λ	0.5	—
	Abstention bonus δ	0.1	—
	Reward function	Full Reward	EM
	Training time		~72 hours

Table 14: Training hyperparameters for PACE-SQL with DeepSeek-Coder-6.7B as the backbone. “—” marks inapplicable settings: the open-source variant uses only R_{match} during RL, as open-source corpora lack user adoption signals, and thus has neither adoption weight λ nor abstention bonus δ .

to align with a line boundary. This trains the model to recover a syntactically complete snippet—such as a full statement or loop block rather than a truncated token slice. The pre-trained weights are publicly released at <https://github.com/aixcoder-plugin/aiXcoder-7B>.

K.2 StructureCoder-7B

StructureCoder (Ren et al., 2025) is trained on the APPS training set (5,000 samples) and validated on Qwen2.5-Coder-Instruct (1.5B / 3B / 7B). It first applies AST-Enhanced FIM, extracting if / for / while / function blocks via the Abstract Syntax Tree as middle segments to build syntactically clean and diverse FIM data. During the subsequent FIM-Enhanced DPO stage, the DPO loss is confined to the middle segment to avoid penalizing correct prefix/suffix tokens, complemented by a length-based curriculum from short to long blocks for progressive difficulty. The pre-trained weights are publicly released at <https://github.com/SenseLLM/StructureCoder>.

K.3 FrugalCoder and Reproduction Details

The other two abstention-related works (Mozannar et al., 2024; de Moor et al., 2026) are built

on proprietary internal codebases and release neither their training code nor their data-processing pipelines, which prevents end-to-end reproduction under a unified setting. We therefore reproduce only FrugalCoder and align it with our method under the same evaluation protocol.

Method. FrugalCoder (Sun et al., 2024) introduces a lightweight *completion quality estimator* into code completion systems. The estimator scores each prompt and rejects it when the score falls below a threshold, avoiding unhelpful suggestions. We reproduce its best variant, **ET**, a small encoder-only Transformer that reuses BERT token embeddings. Their code is released at <https://github.com/v587su/FrugalCoder>.

Reproduction Setup. We follow the original data-construction recipe. Training prompts are sampled from the same data used to train PACE-SQL: WeSQL-1.5M for the internal evaluation and OpenSQL-FIM for the open-source evaluation. From each source, we draw 30K prompts, matching the training-set size used in the original FrugalCoder setup. For every prompt, the target completion model produces a candidate completion, and the BLEU score between this candi-

date and the reference completion serves as the prompt’s quality label. The estimator takes the prompt tokens as input and is trained with a regression objective. All hyperparameters (learning rate, batch size, number of epochs) follow the defaults recommended in the original paper.

Two-Backbone Setup. To ensure a fair comparison, we reproduce FrugalCoder on two backbones, Qwen2.5-Coder-14B and DeepSeek-Coder-6.7B. For each backbone, the candidate completions used to build the training labels are produced by the backbone itself, and the same backbone serves as the downstream completion model, ensuring that the estimator is always matched to its target model. When reporting results, we align the number of retained (non-abstained) suggestions of FrugalCoder with that of PACE-SQL, so that quality metrics such as Exact Match and execution accuracy are not confounded by how many prompts each method chooses to answer.

System Prompt

You are a SQL code completion engine specialized in Fill-in-the-Middle (FIM) tasks.

Task Description

Given a SQL snippet split into two parts — a **prefix** (the SQL code before the missing section) and a **suffix** (the SQL code after the missing section) — your job is to generate the **middle SQL code** that logically and syntactically bridges the prefix and the suffix to form a valid, complete SQL statement.

Input Format

The user will provide input in the following structure:

```
```\n<prefix>\n{SQL code before the missing part}\n</prefix>\n<suffix>\n{SQL code after the missing part}\n</suffix>\n```
```

#### ## Output Format

- Return **ONLY** the middle SQL code that should be inserted between the prefix and the suffix.
- Do NOT include the prefix or suffix in your output.
- Do NOT wrap your output in markdown code fences or any other formatting.
- Do NOT include any explanation, commentary, or extra text.
- The generated middle code must follow correct SQL syntax (e.g., proper use of keywords, clauses, functions, aliases, joins, subqueries, etc.) and seamlessly connect the prefix and suffix into a valid SQL statement.

### User Prompt

```
<prefix>{prefix}</prefix><suffix>{suffix}</suffix>
```

Figure 6: Prompt template used to evaluate closed-source LLMs on SQL FIM tasks.

## System Prompt

You are a reviewer for a SQL Fill-in-the-Middle (FIM) benchmark built from open-source SQL corpora.

Each sample contains:

- prefix p: text left of the cursor
- suffix s: text right of the cursor
- user\_gt m: ground-truth middle

m == "" denotes a non-completion label, meaning no completion should be inserted at the cursor.

Your goal is to determine whether the sample should be kept in the benchmark.

The reviewer should prioritize removing severely noisy or unusable samples while remaining conservative about discarding potentially valid labels.

### TASK FORMALIZATION

label\_type:

- "non\_completion" iff m == ""
- "completion\_required" iff m != ""

Verdict:

- accept: clean and label-consistent
- borderline: usable with moderate uncertainty or weak inconsistencies
- reject: severely noisy, malformed, or clearly label-inconsistent

Only rejected samples are removed.

### STAGE 1 — SEGMENT USABILITY REVIEW

Assess whether p + s forms a usable SQL-centric segment.

A usable segment should:

- be predominantly SQL or SQL-related procedural code
- preserve meaningful cursor context
- avoid severe corruption or unrelated content

Chinese and English comments/string literals are allowed. Limited multilingual text may appear if the surrounding SQL context remains usable.

Reject if any of the following strongly applies:

- G1: malformed, empty, or dominated by non-SQL noise
- G2: little SQL content and no meaningful cursor context
- G3: severe corruption, dumps, stack traces, binary-like content, unrelated programming languages, or repetitive filler
- G4: extensive natural-language content that substantially harms benchmark usability or consistency

Borderline if:

- mostly SQL but with moderate noise
- weak cursor context
- limited multilingual content
- partial corruption that does not destroy usability

Otherwise accept.

If Stage 1 returns reject, skip Stage 2.

### STAGE 2 — LABEL CONSISTENCY REVIEW

Evaluate whether m is reasonably consistent with the cursor position.

The review should remain conservative about rejecting labels. When label correctness is uncertain but still plausible, prefer accept or borderline over reject.

#### 2.1 Cursor-State Inference

Infer the dominant cursor context near the FIM position.

Cursor states:

- c1 = blank\_line: cursor on an empty or whitespace-only line
- c2 = after\_comma: strip(p) ends with ","
- c3 = right\_is\_closer: s begins with a closing token such as ")", "]", "}", quote, or backtick
- c4 = at\_eof: strip(s) is empty
- c5 = glued\_between\_tokens: cursor lies between two adjacent non-whitespace token characters
- c6 = after\_semicolon: strip(p) ends with ";
- c7 = other

If multiple states apply, prefer the one most informative for judging completion plausibility.  
Also consider nearby continuation signals such as incomplete SQL keywords, operators, unmatched open structures, and comment/string boundaries.

---

## 2.2 Non-completion Review (m == "")

---

Accept if abstention is plausible, including cases such as:

- completed statements
- statement boundaries
- cursor before a closing token
- cursor inside comments or strings
- glued token positions
- weak or ambiguous continuation signals

Borderline if continuation appears somewhat likely but still ambiguous.

Reject only if:

- continuation is clearly required
- the cursor strongly implies a missing SQL continuation

---

## 2.3 Completion-required Review (m != "")

---

Accept if p || m || s forms a plausible SQL continuation.

Borderline if:

- continuation is parseable but weakly supported
- completion is unusually long
- mild overlap or dialect inconsistency exists
- local intent appears somewhat mismatched but still usable

Reject only if:

- completion causes severe structural inconsistency
- continuation is clearly unrelated to the surrounding SQL context
- m is effectively empty or non-content

---

## STAGE AGGREGATION

---

Final verdict:

- reject: Stage 1 rejection or clear Stage 2 failure
- borderline: moderate uncertainty without severe failure
- accept: otherwise

keep = (verdict != "reject")

---

## OUTPUT FORMAT

---

Return exactly one JSON object. No prose outside JSON.

```
{
 "noise_gate": "accept|borderline|reject",
 "noise_reasons": ["<G1|G2|G3|G4>", ...],
 "cursor_state": "<c1|c2|c3|c4|c5|c6|c7>",
 "label_type": "non_completion|completion_required",
 "violations": ["<short_reason>", ...],
 "soft_flags": ["<short_reason>", ...],
 "verdict": "accept|borderline|reject",
 "keep": true|false,
 "rationale": "<=50 English words"
}
```

Constraints:

- exactly one JSON object
- no additional keys
- keep == (verdict != "reject")
- noise\_reasons  $\subseteq \{G1, G2, G3, G4\}$

Figure 7: Prompt used for LLM-as-a-judge filtering.