Learning from Near-Misses: Error-Aware Contrastive Few-Shot Learning for NL2Formula

Zhihao Shuai[1*] **Yiyun Chen**[1*] **Maolin Ma**[2] **Yutong Chen**[1]
Hanjia Qiu[2] **Jing Xu**[1] **Ziye Chen**[1,3] **Weikai Yang**[1†]
[1]The Hong Kong University of Science and Technology (Guangzhou)
[2]Southwest University
[3]University College London

Abstract

Natural Language to Excel Formula (NL2Formula) translates user intent into executable spreadsheet formulas. However, current models often produce *near-miss* outputs—formulas that parse correctly yet fail at execution due to an incorrect function, operator, or reference. Through a systematic error analysis, we find that these errors repeatedly arise from a small set of structural decision points, motivating the need for typed error supervision rather than general error signals. To this end, we introduce an abstract syntax tree (AST)-based error taxonomy that organizes common error modes by the kind of decision that goes wrong in the parse tree. Building on this taxonomy, we propose Error-Aware Contrastive Few-Shot Learning (ECFL), an error-aware framework that unifies training and inference around typed error supervision. During offline training, ECFL mines near-miss errors, assigns error types under the taxonomy, and builds error-aware contrastive demonstrations for fine-tuning. During online inference, a lightweight predictor estimates likely error types and triggers targeted retrieval of contrastive demonstrations to guide single-generation inference. Experiments show ECFL improves Exact Match (EM) by 6.4 points over supervised fine-tuning (SFT) and matches self-consistency (SC@5) accuracy at substantially lower inference cost.

1 Introduction

Spreadsheet formulas are a backbone of real-world data analysis and lightweight automation (Chen et al., 2024; Aivaloglou et al., 2017), but writing correct formulas remains a major barrier for non-expert users (Chen et al., 2024; Zhao et al., 2024; Powell et al., 2008). While large language models (LLMs) have lowered the barrier by translating natural language into spreadsheet formulas (Srinivasa Ragavan et al., 2022; Zhao et al., 2024), they often struggle with the strict structural constraints of spreadsheet formulas (Aivaloglou et al., 2017). As a result, models often produce *near-miss* outputs that parse correctly but fail at execution due to a wrong function, operator, or reference. Such near-misses are particularly frustrating because they are usually hard to catch by inspection and can propagate to downstream analyses and decisions (Powell et al., 2008). The long-tail distribution of functions and complex nested compositions further amplify the problem (Chen et al., 2024; Aivaloglou et al., 2017; Zhao et al., 2024), making near-misses both common and costly (Powell et al., 2008).

Through a systematic error analysis (Section 3), we find that these errors are rarely arbitrary but repeatedly arise from a small set of structural decision points in the formula's abstract syntax tree (AST). Existing methods either discard these errors entirely (standard SFT) or collapse them into a single undifferentiated negative signal (e.g., DPO; Rafailov et al., 2023), losing the structural distinctions needed for targeted correction. Self-correction methods iteratively refine outputs by generating a candidate, executing it (or checking syntax), diagnosing errors, and regenerating, but this loop incurs substantial inference cost (Chen et al., 2023; Zhang et al., 2023). Inference-time scaling methods such as self-consistency (Wang et al., 2023) improve accuracy by generating K candidates and selecting via majority vote, but pay a $K\times$ decoding cost. Overall, these methods do not effectively exploit near-miss errors as structured supervision. We therefore hypothesize that typed near-miss supervision can teach type-specific correction patterns that generic preference signals and brute-force sampling do not capture.

To operationalize this idea, we introduce an AST-based *error taxonomy* that types near-misses by the kind of structural choice that goes wrong in the

*Equal contribution.
†Corresponding author.

parse tree. Typing follows a skeleton-first rule that prioritizes structural mismatches in the AST skeleton over leaf-level substitutions (Section 3.3). Only when the skeletons match do we type fill-level errors such as reference or value mismatches. Building on this taxonomy, we propose ECFL (Error-Aware Contrastive Few-Shot Learning), an error-aware framework that exploits typed near-miss supervision for both training and inference. During offline training, ECFL mines near-miss predictions by Monte Carlo sampling, assigns each a deterministic error type via AST comparison, and stores typed wrong–correct pairs in an error bank. For Low-Rank Adaptation (LoRA) fine-tuning, ECFL constructs contrastive prompts by selecting type-aligned demonstrations from the error bank. During online inference, a lightweight predictor estimates likely error types and retrieves one demonstration per estimated type via query-embedding similarity. The model then decodes once, guided by these type-aligned demonstrations with minimal overhead.

To evaluate reliably, we clean a standard benchmark, NL2Formula (Zhao et al., 2024), and introduce TFBench, an expert-curated benchmark covering 215 functions and 12 operators with nested compositions. Experiments show consistent gains in both structural and execution correctness across standard and expert benchmarks, with larger improvements on long-tail functions, while matching or exceeding SC@5 accuracy using single-generation inference (one generation plus a forward-only pass for type prediction), instead of $5\times$ full generations required by SC@5.

Our contributions are threefold:

- **A 9-type near-miss error taxonomy** that provides automatic and interpretable error types for NL2Formula results.
- **ECFL**, an error-aware framework that unifies training and inference around typed near-miss supervision.
- **TFBench**, an expert-curated benchmark and cleaned NL2Formula splits for reliable evaluation. Experiments show consistent improvements over strong baselines on both structural and execution correctness.

2 Related Work

Our work connects several research threads; we summarize key distinctions here and provide an extended discussion in Appendix C.

NL2Formula and Code Generation. Prior systems such as SpreadsheetCoder (Chen et al., 2021b) and FLAME (Joshi et al., 2023) frame formula generation as structured prediction over spreadsheet context, but remain brittle on long-tail functions and complex nesting. These methods rely on end-to-end memorization and lack explicit mechanisms to handle the systematic near-miss errors that arise from strict structural constraints.

Learning from Errors. Self-debugging approaches (Chen et al., 2023; Zhang et al., 2023) use execution feedback to iteratively refine outputs, but require multiple generation-execution cycles. Alternatively, preference learning methods like DPO (Rafailov et al., 2023) leverage collected errors as negative signals to sharpen decision boundaries. In contrast, we *mine* errors offline and type them via AST analysis, converting failure patterns into *error-aware* contrastive supervision for both training and inference, rather than treating all errors as uniform preference signals.

Inference-Time Scaling. Methods like self-consistency (Wang et al., 2023) and Tree of Thoughts (Yao et al., 2023) improve robustness via multi-sample aggregation or deliberate search, but incur prohibitive $K\times$ or more inference cost. Our approach avoids multi-sample generation by predicting likely error types and retrieving targeted contrastive demonstrations with a single extra forward-only pass, yielding comparable robustness at substantially lower cost.

3 Near-Miss Error Taxonomy

3.1 Error Analysis

During the analysis of existing NL2Formula benchmark (Zhao et al., 2024) and the construction of our expert-curated TFBench, annotators wrote gold formulas with model-generated candidates as references. Systematic comparison of these candidates against the final gold formulas revealed two consistent patterns.

First, most errors are *near-misses* rather than complete failures. The model typically understands the task but makes localized errors, such as adding or omitting a formula component, confusing similar functions (e.g., `SUMIF` vs. `COUNTIF`), or selecting the wrong operator (e.g., > vs. >=). While the overall formula structure remains reasonable, the difficulty lies in precise element selection and correct nesting depth.

	Type	Condition	Example
Skeleton	FuncMismatch	$\|S^{\mathrm{w}}\|=\|S^{\star}\|$, diff at func	SUM→AVG
	FuncMissing	$\|S^{\mathrm{w}}\|<\|S^{\star}\|$, missing func	IF(SUM)→SUM
	FuncRedundant	$\|S^{\mathrm{w}}\|>\|S^{\star}\|$, extra func	A1→SUM(A1)
	OpMismatch	$\|S^{\mathrm{w}}\|=\|S^{\star}\|$, diff at op	>→>=
	OpMissing	$\|S^{\mathrm{w}}\|<\|S^{\star}\|$, missing op	A11+B11+C11→A11+B11
	OpRedundant	$\|S^{\mathrm{w}}\|>\|S^{\star}\|$, extra op	A11*B11→A11*B11*2
Fill	RefMismatch	$S^{\mathrm{w}}=S^{\star}$, diff at ref	A1:A10→A1:A5
	ValMismatch	$S^{\mathrm{w}}=S^{\star}$, diff at val	"Yes"→"Y"
	Miscellaneous	Otherwise	COUNT(A:A,"*")→COUNT(A:A)

Table 1: The 9-type error taxonomy with formal conditions. Skeleton errors (top) arise from structural mismatches; fill errors (middle) arise from leaf-level substitutions under identical structure; miscellaneous errors (bottom) capture residual cases that do not fit the structural or fill definitions.

Second, reference and value errors are overwhelmingly *mismatches*, rather than *missing* or *redundant*. Among approximately 10,000 error samples, only 19 cases (<0.2%) involve missing or redundant references and values. This asymmetry suggests that once the model commits to a function or operator, it generally follows the correct argument structure.

To sum up, the main challenge is choosing *which* function/operator to use and *what* reference/value to fill, rather than deciding *how many* arguments to provide. Moreover, errors are typically *structurally localized*: a single term in the formula is wrong while surrounding context remains correct. This locality makes an AST-based method a natural fit for both locating errors and assigning an interpretable error type by deterministic tree comparison. It therefore enables automatic error typing without manual annotation, enabling targeted retrieval of wrong–correct contrastive demonstrations.

3.2 Error Taxonomy

Based on the above analysis, we define a 9-type error taxonomy organized into three levels (Table 1). Figure 1 also illustrates a concrete example. *Skeleton-level* errors (6 types) reflect structural decision failures: function and operator nodes each admit Mismatch, Missing, and Redundant modes. *Fill-level* errors (2 types) occur when the skeleton is identical but leaf content differs, yielding `RefMismatch` and `ValMismatch`. Since fill-level Missing/Redundant errors are negligible (<0.2%), we group them with unparseable cases into *Miscellaneous* (1 type), which is excluded from contrastive demonstration retrieval.

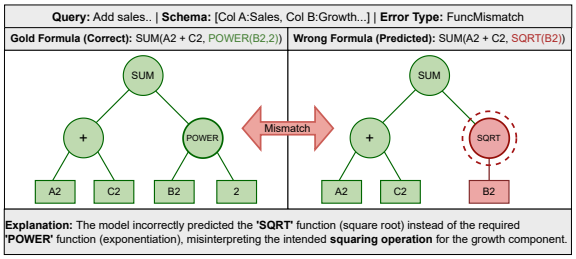

Figure 1: AST-based error typing for formula prediction. The predicted formula incorrectly uses `SQRT` instead of `POWER`, resulting in a `FuncMismatch` error type.

3.3 Error Typing

We formalize error typing as a deterministic function $\tau : (f^{\mathrm{w}}, f^{\star}) \mapsto e$. Given a formula f, its **Abstract Syntax Tree** $\mathcal{T}(f)$ has internal nodes for functions and operators, and leaf nodes for references and values. We briefly review these basic formula components in Appendix H. The **skeleton** $S(f)$ is the sequence of internal node labels obtained by preorder traversal; the **leaf sequence** $L(f)$ is defined analogously for leaves.

Our typing rule follows a **skeleton-first** principle: structural errors take precedence over fill-level errors, as they reflect more fundamental decision failures. Given $S^{\mathrm{w}} = S(f^{\mathrm{w}})$ and $S^{\star} = S(f^{\star})$, we first compare skeleton lengths to determine Missing ($|S^{\mathrm{w}}| < |S^{\star}|$), Redundant ($|S^{\mathrm{w}}| > |S^{\star}|$), or Mismatch (equal length but different node type), then identify whether the first differing node is a function or operator. Only when $S^{\mathrm{w}} = S^{\star}$ do we proceed to leaf comparison, classifying errors as `RefMismatch` or `ValMismatch` based on the first differing leaf type. Cases that do not fit any pattern are assigned `Miscellaneous`.

Since both skeleton and leaf sequence are generated by preorder traversal, left-to-right comparison effectively simulates tree traversal: the first mismatched position corresponds to the first erroneous decision point in the AST. This design aligns error localization with the top-down, left-to-right generation order of autoregressive language models, where the first incorrect token often causes subsequent errors. It is easy to show that this taxonomy is exhaustive (every parseable error receives exactly one type), mutually exclusive (skeleton-first prevents ambiguity), and deterministic (pure AST comparison, no execution or manual annotation). The formal justification of determinism, mutual exclusivity, and exhaustiveness is provided in Appendix I.

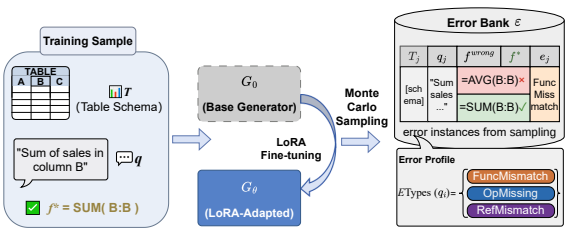

Figure 2: Data structures and notation. A training sample contains table schema T, query q, and gold formula $f^\star$. The error bank $\mathcal{E}$ stores typed error tuples, each pairing a wrong prediction with the correct answer and an AST-derived error label e_j. The error profile $\mathrm{ETypes}(q_i)$ tracks which error types a query triggered during sampling.

4 Method

Building on the error taxonomy defined in Section 3, we now present ECFL, an error-aware framework that transforms near-miss errors into supervisory signals. We organize ECFL into three components: error bank construction, offline training, and online inference (Figure 3).

4.1 Error Bank Construction

We first construct an error bank as a reusable repository of typed near-miss mistakes mined from a base generator $\mathcal{G}_0$ (e.g., an untuned LLM). This error bank supports type-conditioned selection of wrong–correct demonstrations for both offline training and online inference. Each training instance of NL2Formula task consists of a table schema T_i, a natural language query q_i, and a gold formula $f_i^\star$. To mine near-miss errors, we perform Monte Carlo sampling on $\mathcal{G}_0$ by drawing K samples for each (T_i, q_i) under temperature τ and nucleus sampling:

$$f_i^{(k)} \sim p_{\mathcal{G}_0}(f \mid T_i, q_i; \tau, \text{top-}p), \quad k = 1, \ldots, K. \tag{1}$$

We retain an instance only if $f_i^{(k)} \neq f_i^\star$ and its error type is not MISCELLANEOUS. Each retained near-miss is stored in the error bank $\mathcal{E}$ as a typed tuple $(T_i, q_i, f_i^{\text{wrong}}, f_i^\star, e_i)$. We also update the per-query error profile $\mathrm{ETypes}(q_i)$, which records the set of error types observed for q_i during sampling. Figure 2 illustrates these data structures.

4.2 Offline Training

Contrastive LoRA fine-tuning. After constructing the error bank, we build contrastive demonstrations to help the model identify and avoid near-miss errors. For each training query q_i, we select N demonstrations (e.g., N=3) from the error bank $\mathcal{E}$. Each demonstration is formatted as a wrong–correct pair under its table schema, explicitly highlighting the decision boundary (Figure 4). If $\mathrm{ETypes}(q_i)$ is non-empty, we select one demonstration per observed error type by maximizing cosine similarity $\mathrm{Sim}(q_i, q_j)$ between query embeddings (from a sentence encoder like MiniLM (Wang et al., 2020)):

$$d_e = \underset{(T_j, q_j, \cdot, \cdot, e_j) \in \mathcal{E},\, e_j = e,\, j \neq i}{\arg\max} \mathrm{Sim}(q_i, q_j). \tag{2}$$

If fewer than N demonstrations are retrieved, we pad the remaining slots with globally most similar demonstrations. We fine-tune the base generator $\mathcal{G}_0$ with LoRA adapters to obtain $\mathcal{G}_\theta$ on these contrastive demonstrations, minimizing standard autoregressive NLL:

$$\mathcal{L}_{\text{SFT}} = -\sum_{t=1}^{|f_i^\star|} \log p_{\mathcal{G}_\theta}\left(f_{i,t}^\star \mid f_{i,<t}^\star, d_1, \ldots, d_N\right). \tag{3}$$

Error type predictor. To enable error-aware retrieval at inference time (Section 4.3), we need to predict which error types a query is likely to trigger without access to the gold formula. We train a lightweight multilayer perceptron (MLP) classifier $\mathcal{C}$ to estimate error types from the LLM's hidden states. Specifically, for each tuple $(T_i, q_i, f_i^{\text{wrong}}, f_i^\star, e_i)$ in the error bank $\mathcal{E}$, we run a forward-only pass through $\mathcal{G}_\theta$ with (T_i, q_i) and extract the last-token hidden state $\mathbf{h}_i$. Each $(\mathbf{h}_i, e_i)$ pair becomes one training instance. The classifier predicts error types via:

$$p(e \mid q) = \mathrm{softmax}\big(\mathcal{C}(\mathbf{h}_{\text{last}})\big), \tag{4}$$

trained with standard cross-entropy loss. This method achieves 78% macro-F1, verifying the effectiveness of our error-type predictor. Further training and evaluation details are provided in Appendix A.

4.3 Online Inference

Given a test instance (T, q), we first run a forward-only pass through $\mathcal{G}_\theta$ to obtain the hidden state $\mathbf{h}$. The classifier $\mathcal{C}$ outputs a distribution over error types, from which we select the top-M types $\hat{E} = \{\hat{e}_1, \ldots, \hat{e}_M\}$. For each $\hat{e}_m \in \hat{E}$, we retrieve the most similar demonstration:

$$d_{\hat{e}_m} = \underset{(T_j, q_j, \cdot, \cdot, e_j) \in \mathcal{E},\, e_j = \hat{e}_m}{\arg\max} \mathrm{Sim}(q, q_j). \tag{5}$$

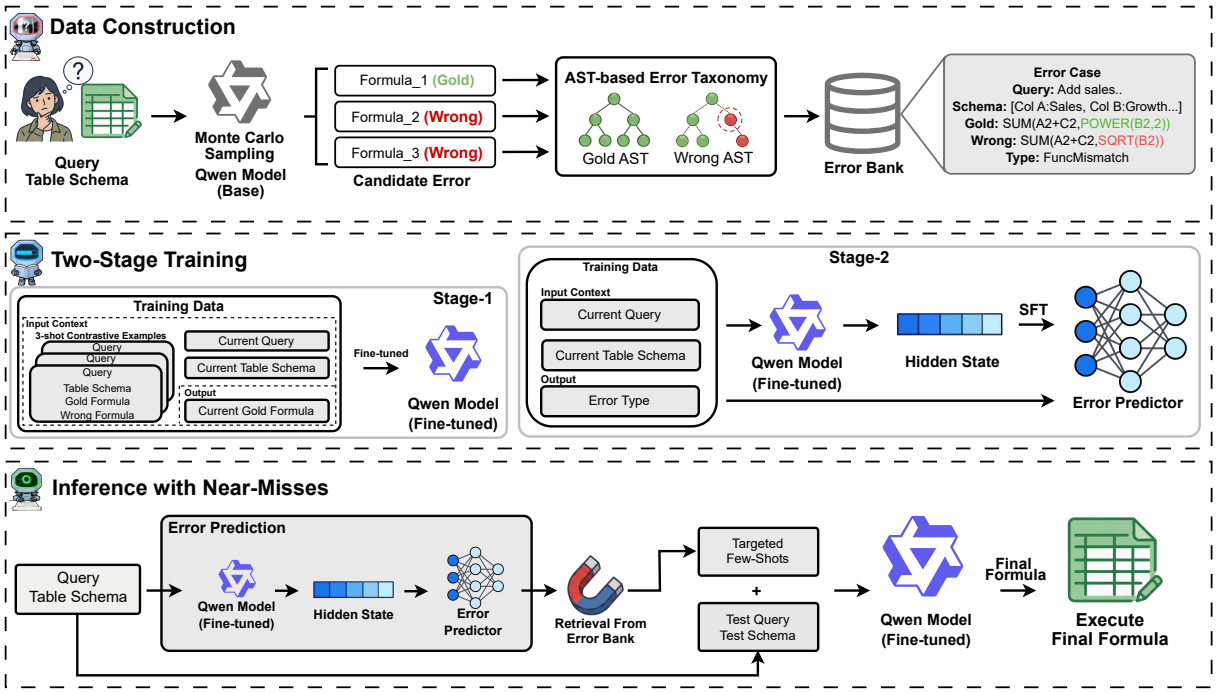

Figure 3: Framework overview. We first build an error bank by Monte Carlo sampling and assign AST-based types into 9 categories (Section 3). The error bank is used to construct error-aware contrastive demonstrations for LoRA fine-tuning, and train an error-type predictor. During inference, we predict error types, retrieve type-aligned demonstrations, and generate the final formula.

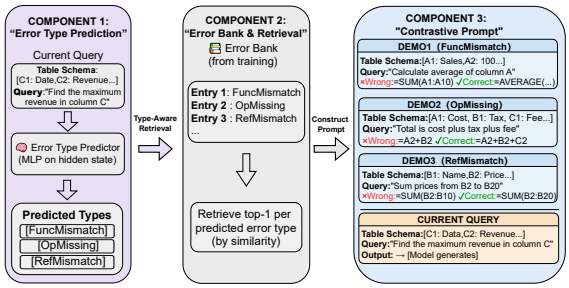

Figure 4: Error-aware contrastive prompt construction. Each demonstration shows a wrong–correct pair under its table schema, followed by the current query.

The retrieved demonstrations form a contrastive prompt, and $\mathcal{G}_\theta$ generates the final formula $\hat{f}$ in a single pass.

This *predict–retrieve–generate* procedure adds minimal overhead (one forward pass for hidden states, plus fast retrieval) while providing targeted contrastive evidence aligned with the query's likely failure modes. The classifier is a small MLP over one hidden state and the retrieval step is a single similarity lookup, so their cost is negligible compared to full autoregressive decoding of a formula.

Algorithm summary. Algorithm 1 summarizes the complete ECFL framework.

5 Experiments

We evaluate ECFL on two benchmarks and compare against different training-based methods, inference-time scaling methods, and API baselines.

5.1 Experimental Setup

5.1.1 Datasets

NL2Formula (public). We use NL2Formula (Zhao et al., 2024) as the in-domain benchmark. Public NL2Formula corpora contain substantial noise, so we systematically audit the data and remove low-quality samples, filtering out 7k+ problematic instances in total: (i) *domain contamination*—formulas mixing Excel with Power BI/DAX-specific syntax (e.g., `SUMMARIZE`, `SUMX`); (ii) *redundant wrappers*—formulas that wrap dynamic-array expressions with `SUM()` to force a single scalar value; (iii) *annotation mismatches*—samples where the natural language question does not align with the formula. We split the cleaned data into training / test / validation sets with a near 7:2:1 ratio in a cross-table manner, ensuring no table appears in more than one split. All baselines are trained and evaluated on this cleaned data, ensuring fair comparison.

Algorithm 1 ECFL

Require: Data $\mathcal{D}$, base generator $\mathcal{G}_0$
Ensure: Generator $\mathcal{G}_\theta$, classifier $\mathcal{C}$, error bank $\mathcal{E}$

```
/* Phase 1: Offline Training */
E ← ∅
for (T_i, q_i, f_i*) ∈ D do
    for k = 1 to K do
        f_i^(k) ~ p_{G_0}(f | T_i, q_i)
        if f_i^(k) ≠ f_i* and parseable then
            e ← AstType(f_i^(k), f_i*)
            if e ≠ Miscellaneous then
                E ← E ∪ {(T_i, q_i, f_i^(k), f_i*, e)}
            end if
        end if
    end for
end for
for (T_i, q_i, f_i*) ∈ D do
    D_i ← ErrorAwareRetrieve(q_i, E, ETypes(q_i))
    Fine-tune G_θ with contrastive prompt from D_i
end for
Train C on {(h_i, e)} pairs from E

/* Phase 2: Online Inference */
Input: Test instance (T, q)
h ← G_θ.Forward(T, q)
Ê ← TopK(C(h), N)
D_demo ← ErrorAwareRetrieve(q, E, Ê)
f̂ ← G_θ.Generate(T, q, D_demo)
return f̂
```

TFBench (expert-curated). Existing benchmarks over-represent common functions like `SUM` and `IF`, which can mask brittleness on long-tail functions and complex compositions. To stress-test compositional generalization on long-tail functions, we construct TFBench from publicly sourced tables and expert-annotated gold formulas. TFBench includes 41k+ expert-annotated instances, covering 215 distinct Excel functions and 12 operators. The functions include rarely used configurations (e.g., advanced `XLOOKUP` parameters) and newer constructs (`LET`, `LAMBDA`). These instances are stratified by nesting depth (1–11) and function rarity, enabling fine-grained analysis of where models fail. Figure 5 illustrates function co-occurrence patterns, showing how functions commonly nest together in complex formulas. Additional statistics and annotation details are provided in Appendix D and Appendix E.

5.1.2 Baselines

We compare ECFL against baselines spanning four categories (details in Appendix B):

Training-based methods. We consider both standard SFT and its variant with preference optimization. SFT fine-tunes the base LLM on cleaned NL2Formula with standard instruction format.

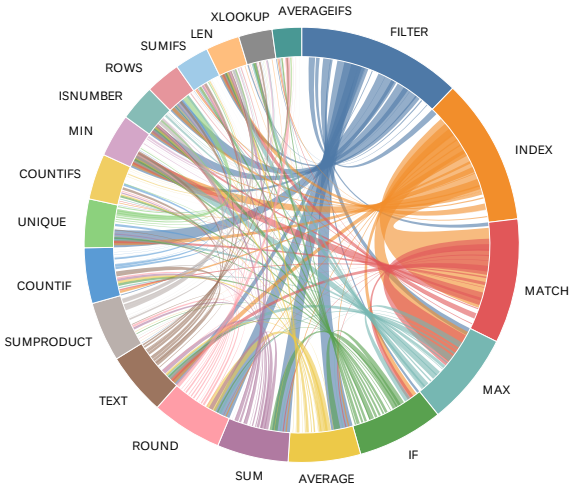

Figure 5: Chord diagram of function co-occurrence in TFBench. The 20 most frequent functions are arranged around the circle, with arc size proportional to frequency. Ribbons indicate co-occurrence within the same formula.

SFT + DPO (Rafailov et al., 2023) further applies preference optimization using query-specific wrong–correct preference pairs constructed from our mined near-miss errors. Concretely, for each training query, we use type-matched near-miss outputs from the error bank as rejected responses and the corresponding gold formula as the chosen response.

Inference-time scaling methods. These methods build on the SFT model and improve accuracy by inference-time scaling. Self-Consistency (Wang et al., 2023) (SC@k) samples k outputs and selects via majority vote. Self-Debugging (Chen et al., 2023) iteratively refines outputs based on execution feedback.

In-Context Learning (ICL). This method retrieves the most similar query-gold exemplars from the training set via embedding similarity, uses them as demonstrations for both training and inference, but without error typing or wrong–correct contrastive pairs. This tests whether retrieval alone suffices, or whether contrastive pairs are necessary.

API LLMs. We evaluate DeepSeek-R1, Gemini-2.5-Pro, Claude-3.7-Sonnet, and GPT-4o under zero-shot settings.

5.1.3 Metrics

We report two complementary metrics. **Exact Match (EM@1)** normalizes formulas by removing redundant whitespace, canonicalizing function-

Component	Setting
MC samples	K=3, τ=0.7, top-p=0.9
Demos	N=3
Predicted types	M=3 (top-3 from MLP)
Encoder	MiniLM (cosine sim.)
LoRA rank/α/dropout	32/64/0.05
LR/Batch/Epoch	2e-5/4/3
Max length	2500
MLP layer	4

Table 2: Key hyperparameters of our method.

name case, and standardizing separators, then computes an exact string match against the gold formula. This metric is fully reproducible and aligns with our training-time verification. **Execution Accuracy (Exec@1)**, when a spreadsheet execution engine and input tables are available, counts a prediction as correct if it produces the same output as the gold formula on the provided table.

5.1.4 Implementation

We use Qwen3-8B as the base LLM ($\mathcal{G}_0$) and adapt it with LoRA to obtain $\mathcal{G}_\theta$ (Section 4). For Monte Carlo error mining, we sample K=3 candidates per query with τ=0.7 and top-p=0.9. For retrieval, we embed queries using MiniLM and compute cosine similarity. Error-aware retrieval fetches the most similar demonstration per selected error type. Key hyperparameters are summarized in Table 2.

For API LLMs, we use identical input format (table schema + query) and enforce a strict output format ("Return *only* the Excel formula"). We report results under zero-shot prompting. Unless otherwise stated, temperature is set to 0 (or the closest deterministic setting) to reduce variance, with output length capped to avoid verbose explanations.

5.2 Results

Table 3 reports results on NL2Formula and TFBench. On NL2Formula, ECFL performs the best and achieves the highest EM (75.32%) and Exec (86.24%), improving over SFT by +6.4 EM and +6.1 Exec. SFT+DPO improves EM over SFT but reduces Exec (80.18% $\rightarrow$ 79.01%), suggesting overfitting to surface formatting. In comparison, our error-aware contrastive supervision provides a more reliable learning signal. On TFBench, ECFL improves over SFT from 38.46% to 42.05% EM (+3.6) and from 58.91% to 61.25% Exec (+2.3). The gain is larger on rare functions (+16.28 EM). This suggests that ECFL is more effective on long-tail functions. Fine-grained analyses are provided in Appendix F.

	NL2Formula		TFBench	
Method	**EM**	**Exec**	**EM**	**Exec**
Training Methods				
SFT	0.6894	0.8018	0.3846	0.5891
SFT + DPO	0.7006	0.7901	0.3876	0.5626
ECFL (Ours)	**0.7532**	**0.8624**	**0.4205**	**0.6125**
Inference-Time Enhancement				
SFT + SC@3	0.7036	0.8162	0.3897	0.5917
SFT + SC@5	0.7169	0.8319	0.4073	0.6095
SFT + Self-Debugging	0.7017	0.8007	0.4020	0.5798
ICL Baselines				
ICL (positive-only)	0.7254	0.8385	0.3948	0.5611
API LLMs				
DeepSeek-R1	0.3318	0.6103	0.2292	0.5456
Gemini-2.5-Pro	0.3497	0.6819	0.2841	0.6025
Claude-3.7-Sonnet	0.3439	0.6122	0.2440	0.5816
GPT-4o	0.3402	0.6612	0.2771	0.5924

Table 3: Main results on NL2Formula and TFBench. EM = Exact Match, Exec = Execution Accuracy.

Compared to inference-time scaling, ECFL outperforms SC@5 on both NL2Formula (75.32% vs. 71.69% EM) and TFBench (42.05% vs. 40.73% EM), while avoiding multi-sample decoding. Self-Debugging underperforms SC@5 and ECFL on EM in both benchmarks, because near-miss formula errors are often silent, limiting the utility of iterative refinement.

ICL (positive-only) is a strong baseline (72.54% EM and 83.85% Exec on NL2Formula), showing that retrieval alone provides substantial gains. However, ECFL remains higher by +2.8 EM and +2.4 Exec on NL2Formula, indicating that typed wrong–correct demonstrations add value beyond query–gold demonstrations alone.

API LLMs exhibit a large EM–Exec gap (e.g., GPT-4o: 34.02% EM vs. 66.12% Exec), suggesting frequent non-canonical yet executable outputs. Yet, ECFL still outperforms API LLMs, and the smaller gap between EM and Exec suggests that contrastive fine-tuning teaches canonical solutions matching both functionally and formally.

5.3 Ablation

Training vs. Inference Contribution. Table 4 ablates whether contrastive demonstrations are used during training (**Train**) and inference (**Infer**). Row 1 uses no contrastive demonstrations in either stage (SFT). Row 2 uses contrastive demonstrations only at inference time. Row 3 uses contrastive demonstrations only during training. Row 4 uses contrastive demonstrations in both stages (full

Train	Infer	NL2Formula		TFBench	
		EM	Exec	EM	Exec
✗	✗	0.6894	0.8018	0.3846	0.5891
✗	✓	0.6271	0.7829	0.3615	0.4916
✓	✗	0.6852	0.7998	0.3940	0.5467
✓	✓	**0.7532**	**0.8624**	**0.4205**	**0.6125**

Table 4: Ablation of training- and inference-time contributions. The Train/Infer columns indicate whether contrastive demonstrations are used, respectively.

Retrieval	NL2Formula		TFBench	
	EM	Exec	EM	Exec
Sim-only (no types)	0.6862	0.7982	0.3914	0.5469
Fixed-types (top-3 types)	0.7092	0.8126	0.4028	0.5567
Predicted-types (ECFL)	**0.7532**	**0.8624**	**0.4205**	**0.6125**

Table 5: Ablation of the usage of error types for contrastive demonstration retrieval.

ECFL). Using both yields the best results (75.32% EM on NL2Formula), while using only training (68.52%) or only inference (62.71%) is substantially worse. Notably, inference-only contrastive demonstrations underperform the SFT baseline, indicating that the model must learn how to use contrastive evidence during training.

Value of Error-Aware Retrieval. Table 5 compares three contrastive demonstrations retrieval strategies. **Sim-only** ignores error types and retrieves demonstrations purely by query similarity. **Fixed-types** always retrieves demonstrations from the three most frequent error types. **Predicted-types** (ECFL) predicts likely error types for each query and retrieves one contrastive demonstration per predicted type. The results show that both fixed-types and predicted-types outperform sim-only, indicating that sampling demonstrations by error type is beneficial. Predicted-types further achieves the best performance, suggesting that selecting error types conditioned on the query is more effective.

Additional results on cross-family generalization and black-box-compatible predictor variants are reported in Appendix G.

6 Conclusion

We propose ECFL (Error-Aware Contrastive Few-Shot Learning), an error-aware training-inference framework that explicitly learns from model mistakes. By mining errors via Monte Carlo sampling, typing them with AST rules, and retrieving error-aware contrastive demonstrations, ECFL improves structural correctness on both the cleaned NL2Formula benchmark and our expert-curated TFBench, exceeding SC@5 accuracy at substantially lower inference cost. These results suggest that *targeted error correction* through typed retrieval offers a compelling alternative to blind memorization or costly multi-sample decoding. We believe this error-aware paradigm generalizes beyond spreadsheets: any structure-sensitive code generation task—where near-misses are systematic and AST-comparable—could benefit from similar typed supervision. We hope ECFL and TFBench serve as useful resources for the community. Our code and data are publicly available at https://github.com/Bvivib-shuai/ECFL.

Limitations

We acknowledge several limitations and outline directions for future work.

Error bank specificity. Our error bank is constructed offline from Qwen3-8B under specific sampling conditions. The optimal bank may differ across model families and deployment settings. In the future, we will explore adaptive bank construction that updates using target-model feedback or online error accumulation.

Predictor accuracy. Our ablation shows that retrieving demonstrations based on query-specific predicted error types improves performance. While our error-type predictor achieves 78% macro-F1, misclassifications still occur. These errors can route retrieval toward less relevant demonstrations. It is beneficial to incorporate uncertainty-aware prediction that abstains or retrieves broader demonstrations when confidence is low, or more generally improve the predictor to better support retrieval.

Taxonomy assumption. Our taxonomy is defined based on the error patterns observed in our error analysis. Due to resource constraints, we did not conduct a dedicated ablation to systematically compare alternative taxonomies or identify an optimal one. As a result, the current taxonomy may miss finer-grained failure modes. Moreover, we currently consider only the first error, which may underrepresent multi-step failures within a single formula. We will explore taxonomy design choices (e.g., granularity, hierarchy, and multi-label typing) and evaluate their impact on demonstration retrieval and formula generation.

Ethical Considerations

Our work focuses on translating natural language to spreadsheet formulas, a low-risk application domain. The datasets used (NL2Formula and TF-Bench) contain only formula expressions and table schemas, with no personally identifiable information or offensive content. We do not foresee direct risks from misuse, as the generated formulas operate within spreadsheet environments with limited external impact.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grants 62502413.

References

Efthimia Aivaloglou, David Hoepelman, and Felienne Hermans. 2017. Parsing excel formulas: A grammar and its application on 4 large datasets. *Journal of Software: Evolution and Process*, 29(12):e1895.

Mark Chen, Jerry Tworek, Heewoo Jun, and 1 others. 2021a. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Sibei Chen, Yeye He, Weiwei Cui, Ju Fan, Song Ge, Haidong Zhang, Dongmei Zhang, and Surajit Chaudhuri. 2024. Auto-Formula: Recommend formulas in spreadsheets using contrastive learning for table representations. *Proceedings of the ACM on Management of Data*, 2(3).

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. In *arXiv preprint arXiv:2304.05128*.

Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. 2021b. SpreadsheetCoder: Formula prediction from semi-structured context. In *International Conference on Machine Learning*, pages 1661–1672. PMLR.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6894–6910.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Harshit Joshi, Chunning Zhao, Bing Li, Yu Liu, and 1 others. 2023. FLAME: A small language model for spreadsheet formulas. *arXiv preprint arXiv:2301.13779*.

Ansong Ni, Srini Iyer, Dragomir Radev, and 1 others. 2023. LEVER: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128.

Stephen G. Powell, Kenneth R. Baker, and Barry Lawson. 2008. A critical review of the literature on spreadsheet errors. *Decision Support Systems*, 46(1):128–138.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, and 1 others. 2023. Code Llama: Open foundation models for code. In *arXiv preprint arXiv:2308.12950*.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546.

Sruti Srinivasa Ragavan, Zhitao Hou, Yun Wang, Andrew D. Gordon, Haidong Zhang, and Dongmei Zhang. 2022. GridBook: Natural language formulas for the spreadsheet grid. In *Proceedings of the 27th International Conference on Intelligent User Interfaces (IUI '22)*, pages 345–368, New York, NY, USA. Association for Computing Machinery.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in neural information processing systems*, 33:5776–5788.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Kechi Zhang, Zhuo Li, Jia Li, and 1 others. 2023. Self-Edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, pages 769–787.

Wei Zhao, Zhitao Hou, Siyuan Wu, Yan Gao, Haoyu Dong, Yao Wan, Hongyu Zhang, Yulei Sui, and Haidong Zhang. 2024. NL2Formula: Generating spreadsheet formulas from natural language queries. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 2377–2388, St. Julian's, Malta. Association for Computational Linguistics.

A Implementation Details

Monte Carlo Sampling. We use temperature $\tau = 0.7$, top-$p = 0.9$, and maximum length 512 tokens, with each query sampled $K = 3$ times.

Error-Type Predictor. The MLP classifier has 4096-dim input $\rightarrow$ 1024-dim hidden (with residual connection) $\rightarrow$ 512-dim hidden $\rightarrow$ 9-dim output, trained with the AdamW optimizer (lr=1e-3, batch size 12 and 100 epochs).

Software Versions. Python 3.10, PyTorch 2.9.1, Transformers 4.57.1, PEFT 0.18.0, Sentence-Transformers 5.2.0.

Reproducibility. All results are from single runs with a fixed random seed (42). We verified stability on the SFT baseline with 3 independent runs, observing standard deviation $<$0.3% on both EM and Exec.

Computational Resources. We use Qwen3-8B (8B parameters) as the base model. All experiments were conducted on 4 NVIDIA A800 80GB GPUs. LoRA fine-tuning takes approximately 3 hours; Monte Carlo error mining takes approximately 2 hours.

Licenses. We use the following open-source resources: Qwen3-8B (Apache 2.0), MiniLM (MIT), NL2Formula dataset (CC-BY 4.0). TF-Bench is constructed from public spreadsheet data sourced from Kaggle (`https://www.kaggle.com/datasets/`) and Alibaba Tianchi (`https://tianchi.aliyun.com/dataset`), which are released under ODbL v1.0, DbCL v1.0, and CC0 1.0 licenses. TFBench will be released under CC-BY 4.0. Our code will be released under MIT license upon acceptance.

Use of AI Assistants. We used AI writing assistants (e.g., ChatGPT, Claude) for grammar checking and formatting corrections during manuscript preparation. All scientific claims, experimental design, methodology, and core contributions are the authors' original work.

B Baseline Details

We describe each baseline category in detail.

Training-based Methods.

- SFT (Supervised Fine-Tuning): Fine-tunes $\mathcal{G}_0$ on cleaned NL2Formula data with standard instruction format. No contrastive demonstrations are used during training or inference.
- SFT + DPO: After SFT, we apply Direct Preference Optimization (Rafailov et al., 2023) using wrong–correct pairs as preference data. This tests whether preference learning can implicitly leverage error signals without explicit error typing.

Inference-Time Scaling Methods.

- SFT + SC@k: Self-Consistency (Wang et al., 2023) samples k outputs from the SFT model and selects the most frequent one via majority vote. We report $k \in \{3, 5\}$.
- SFT + Self-Debugging: Following Chen et al. (2023), the model iteratively refines its output based on execution feedback or syntax errors. We allow up to 3 refinement rounds.

ICL Baselines.

- ICL (positive-only): Retrieves top-N most similar query-gold exemplars from the training set based on query embedding similarity.

API LLMs. We evaluate four strong general-purpose models: DeepSeek-R1, Gemini-2.5-Pro, Claude-3.7-Sonnet, and GPT-4o. Each is tested under (i) zero-shot (schema + query only). We use identical input serialization and enforce a strict output format.

C Extended Related Work

This appendix provides an extended discussion of related work, elaborating on the connections and distinctions summarized in Section 2.

C.1 NL2Formula and Code Generation

The task of generating code from natural language has seen remarkable progress with the advent of large language models. Codex (Chen et al., 2021a) demonstrated that LLMs pretrained on code can achieve strong performance on programming benchmarks. Subsequent work has explored specialized architectures such as CodeT5 (Wang et al., 2021) for code understanding and Code Llama (Roziere et al., 2023) for code-specific tasks.

NL2Formula represents a specialized instance of code generation where the target language

is Excel's formula syntax. SpreadsheetCoder (Chen et al., 2021b) introduced neural encoder-decoder models that incorporate spreadsheet context (row/column headers, cell values) as structured input. FLAME (Joshi et al., 2023) explored smaller, task-specialized models for formula generation. Despite progress, these systems struggle with the strict structural constraints of Excel formulas, where minor errors in function names, argument order, or cell references can completely invalidate the output. Crucially, existing methods rely on end-to-end memorization and lack explicit mechanisms to handle the systematic *near-miss* errors that arise from long-tail function distributions and complex nesting patterns.

C.2 Learning from Model Errors

A growing body of work explores how models can learn from their own mistakes, which is central to our approach.

Self-Debugging and Refinement. Self-debugging approaches use execution feedback to iteratively refine outputs. Chen et al. (2023) teach models to self-debug by generating and executing test cases. Self-Edit (Zhang et al., 2023) trains fault-aware editors to localize and fix errors. LEVER (Ni et al., 2023) learns verifiers that predict execution correctness to filter candidates. These methods typically operate at inference time and require multiple generation-execution cycles. Our approach differs by mining errors *offline* during training, building a reusable error bank that enables single-pass generation at inference time.

Preference Optimization and Error Signals. Direct Preference Optimization (DPO) (Rafailov et al., 2023) and its variants have emerged as powerful alternatives to RLHF for aligning language models with human or model-generated preferences. In the context of code generation, collected errors can serve as the "rejected" samples to guide the model away from failure modes. However, standard DPO treats each error as a monolithic failure, potentially missing the fine-grained structural reasons behind a near-miss. Our framework complements preference-based methods by explicitly typing errors via AST analysis, enabling more granular contrastive supervision during both fine-tuning and retrieval.

Error Typing and Categorization. Prior work on error analysis in code generation often treats errors as monolithic failures. We introduce a fine-grained AST-based error taxonomy that categorizes near-misses into 9 types (function/operator mismatch, missing, redundant, fill-level errors, and miscellaneous). This typed supervision enables targeted retrieval of contrastive demonstrations aligned with predicted error patterns.

C.3 Inference-Time Robustness

For code generation, inference-time strategies can substantially improve robustness but often incur high computational cost.

Self-Consistency. Self-consistency (Wang et al., 2023) samples multiple outputs and selects by majority vote or execution agreement. While effective, sampling K candidates requires K forward passes, leading to multiplicative inference cost. Our approach achieves comparable robustness without multi-sample generation by predicting likely error types and retrieving targeted demonstrations with a single extra forward-only pass.

Execution-Guided Methods. Execution-guided methods (Shi et al., 2022) use program execution to filter or rank candidates. These require access to an execution environment at inference time, which may not always be available. Our error-type predictor anticipates likely failure modes *before* generation, enabling proactive rather than reactive error handling.

C.4 Contrastive Learning for Generation

Contrastive learning has proven effective across NLP tasks by learning representations that distinguish positive pairs from negative pairs (Gao et al., 2021). In the context of generation, contrastive signals can sharpen decision boundaries between correct and incorrect outputs.

We adopt a simple but effective form of contrast: presenting wrong–correct pairs as in-context demonstrations. Unlike representation-level contrastive learning, our approach operates at the output level, directly showing the model examples of common mistakes and their corrections. This interpretable contrast is naturally aligned with the structure of formula generation errors, where near-misses differ from gold outputs by specific, identifiable deviations.

C.5 Parameter-Efficient Fine-Tuning

Adapting large language models to specialized tasks traditionally required full fine-tuning, which is computationally expensive and risks catastrophic forgetting. LoRA (Hu et al., 2022) introduces low-rank adapter matrices that can be efficiently trained

and merged with pretrained weights.

In our framework, LoRA serves as the adaptation mechanism, but the key innovation lies in *what* we fine-tune on: contrastive few-shot prompts constructed from the error bank. By exposing the model to wrong–correct pairs during fine-tuning, we inject error awareness directly into the adapted parameters, complementing inference-time retrieval.

C.6 Data Quality and Benchmark Design

The quality of training data significantly impacts code generation performance. For NL2Formula specifically, public datasets often contain non-Excel functions (from Power BI/DAX), redundant wrapper patterns, and inconsistent annotations. Our data governance pipeline addresses these issues through systematic cleaning: removing domain-mismatched samples, normalizing redundant structures via AST analysis, and canonicalizing references.

Existing NL2Formula benchmarks tend to over-emphasize common functions like `SUM` and `IF`, making it difficult to diagnose failures on rare functions and complex compositions. TFBench addresses this gap by deliberately covering 215 distinct functions and 12 operators, including long-tail functions and advanced constructs (`XLOOKUP`, `LET`, `LAMBDA`), with controlled difficulty stratification by nesting depth. This enables more nuanced evaluation of compositional generalization.

D TFBench Statistics

We construct TFBench by curating public spreadsheet tables sourced from Kaggle and Alibaba Tianchi. The experts are asked to write the corresponding Excel formula using standard Excel syntax. TFBench contains 41k+ expert-annotated instances and is stratified by nesting depth and function rarity for fine-grained analysis (Table 6). The annotation protocol and quality control procedures are described in Appendix E.

Statistic	Value
Total examples	41437
Unique functions	215
Unique operators	12
Avg. nesting depth	2.35
Max nesting depth	11
Avg. formula length	23.82 tokens

Table 6: TFBench statistics.

E TFBench Annotation Protocol

Annotator Instructions. Annotators were instructed to construct both the natural language query and the corresponding Excel formula based on each table. Specifically, given a table schema, annotators first formulated a realistic natural language query describing a spreadsheet task, and then independently wrote the corresponding Excel formula using standard Excel syntax. This process ensures that queries are naturally aligned with table semantics rather than being reverse-engineered from formulas. Finally, they verified the correctness of the formula by mental execution against the table schema. We did not provide step-by-step templates to avoid biasing annotators toward specific function choices.

Quality Control. Each formula was verified by a second annotator. Disagreements were resolved by a third senior annotator.

Annotator Recruitment and Compensation. Annotators were graduate students in mathematics and computer science with proficiency in spreadsheet applications, recruited from the research team. They were compensated at standard research assistant rates. All participation was voluntary, and annotators consented to their annotations being used for research purposes.

Annotator Demographics. Annotation was performed by 10 graduate students (6 male, 4 female) with 2+ years of Excel experience, located in China. No personally identifiable information about annotators is disclosed beyond aggregate demographics.

F Fine-Grained Analysis

F.1 Long-Tail Function Performance

To understand where ECFL gains come from, we stratify TFBench results by function frequency in the training set. Table 7 reports performance on queries containing low-frequency functions.

Thresh.	#Func	#Samp.	SFT	ECFL	Δ
Bot. 0.1%	39	86	20.93	37.21	+16.28
Bot. 0.05%	29	49	12.24	26.53	+14.29

Table 7: Performance on low-frequency functions (%). ECFL shows larger gains on rarer functions.

The results confirm that ECFL is particularly effective on long-tail functions, where error-aware

contrastive demonstrations provide targeted guidance that standard SFT lacks.

F.2 Case Study

Table 8 shows examples where SFT produces incorrect formulas but ECFL succeeds.

Case 1: Calculate the Kurtosis of the trading Volume distribution for the year 2015.
Gold: `=KURT(FILTER(G2:G15097, YEAR(A2:A15097)=2015))`
SFT: `=STDEV(...)` ✗
ECFL: `=KURT(...)` ✓
Error: `FuncMismatch`
Case 2: Forecast the user count when the temperature is 0.5.
Gold: `=FORECAST(0.5, P2:P732, J2:J732)`
SFT: `=FORECAST.ETS(...)` ✗
ECFL: `=FORECAST(...)` ✓
Error: `FuncMismatch`

Table 8: Case study: SFT errors corrected by ECFL.

G Additional Experimental Results

G.1 Cross-Family Generalization and Cross-Model Bank Transfer

To assess whether ECFL generalizes beyond a single backbone family, we additionally evaluate it on Llama-3.1-8B-Instruct and test cross-model bank transfer by applying a Llama-mined error bank to Qwen3-8B. As shown in Table 9, ECFL improves over SFT on Llama-3.1-8B-Instruct across both benchmarks, indicating that the benefit of typed near-miss supervision is not limited to a single model family. Moreover, the transferred-bank setting remains clearly above Qwen3-8B SFT while still below in-family ECFL, suggesting that mined error patterns are partly transferable across families, although model-specific error priors still matter.

G.2 Black-Box Compatibility and Predictor Robustness

To assess whether typed retrieval remains useful without hidden-state access, we additionally evaluate a black-box predictor based on query embeddings and a confidence-aware hybrid fallback strategy. The query-embedding predictor replaces hidden-state features with SentenceTransformer representations of the input query, while the hybrid predictor uses entropy as an uncertainty signal and falls back to similarity-only retrieval when the predicted type distribution is uncertain.

As shown in Table 10, the query-embedding predictor remains competitive, suggesting that typed retrieval is still useful in a black-box-compatible setting. Moreover, the entropy-gated hybrid predictor achieves the best performance on both benchmarks, indicating that confidence-aware fallback can further improve robustness when error-type prediction is uncertain.

H Excel Formula Primer

Excel formulas are tree-structured expressions that combine *functions*, *operators*, *references*, and *values*. In our setting, these four components are the basic units that underlie both formula generation and error typing.

Functions. Functions perform spreadsheet operations and are typically written as a function name followed by one or more arguments in parentheses, such as `SUM(A2:A10)`, `AVERAGE(B2:B10)`, or `XLOOKUP(E2, A2:A20, B2:B20)`. Functions may be nested, meaning that one function can take another function call as an argument.

Operators. Operators combine or compare expressions. Common arithmetic operators include `+`, `-`, `*`, and `/`; comparison operators include `>`, `>=`, `<`, `<=`, and `=`. For example, in `A2+B2`, the symbol `+` is an operator, and in `B2>100`, the symbol `>` is a comparison operator.

References. References point to spreadsheet cells or ranges. A single-cell reference such as `A2` refers to one cell, while a range such as `A2:A10` refers to multiple cells. References may also span multiple columns or rows depending on the formula.

Values. Values are literal constants appearing in formulas, such as numbers (e.g., `100`, `0.5`) or strings (e.g., `"Yes"`). Values often serve as thresholds, conditions, or lookup keys.

Illustrative example. Consider the formula

`=SUMIF(B2:B10, ">100", A2:A10).`

Here, `SUMIF` is a function, `B2:B10` and `A2:A10` are references, and `">100"` contains a comparison operator (`>`) and a literal value (`100`). Intuitively, this formula sums the values in `A2:A10` for rows whose corresponding entries in `B2:B10` are greater than 100.

This decomposition aligns with Section 3: function/operator mistakes are skeleton-level errors,

Backbone	Setting	NL2Formula EM	NL2Formula Exec	TFBench EM	TFBench Exec
Qwen3-8B	SFT	0.6894	0.8018	0.3846	0.5891
	ECFL (In-Family Bank)	**0.7532**	**0.8624**	**0.4205**	**0.6125**
	ECFL (Transferred Bank)	0.7435	0.8408	0.4093	0.5926
Llama-3.1-8B-Instruct	SFT	0.7124	0.7961	0.3794	0.5903
	ECFL (In-Family Bank)	0.7388	0.8207	0.4105	0.6017

Table 9: Cross-family generalization and cross-model bank transfer. "In-Family Bank" denotes an error bank mined from the same backbone model, while "Transferred Bank" denotes an error bank transferred from another family.

Setting	NL2Formula EM	NL2Formula Exec	TFBench EM	TFBench Exec
SFT	0.6894	0.8018	0.3846	0.5891
Query-Embedding Predictor	0.7359	0.8421	0.3908	0.5993
Hidden-State Predictor	0.7532	0.8624	0.4205	0.6125
Hybrid Predictor (Entropy-Gated)	**0.7683**	**0.8716**	**0.4286**	**0.6175**

Table 10: Black-box compatibility and predictor robustness (EM/Exec). The query-embedding predictor provides a black-box-compatible alternative to hidden-state prediction, while the entropy-gated hybrid predictor further improves robustness by falling back to similarity-only retrieval for uncertain cases. The entropy threshold is set to $\tau = 1.462$ based on the validation split and fixed at test time.

while reference and value substitutions are fill-level errors.

I Formal Properties of the Error Taxonomy

In Section 3.3, we describe our error typing rule as a deterministic mapping from a predicted formula and a gold formula to a single error type. We briefly justify three properties of this mapping below: determinism, mutual exclusivity, and exhaustiveness.

Let $\tau(f^w, f^*)$ denote the error type assigned to a predicted formula f^w and a gold formula f^*. For a parseable formula f, let $T(f)$ be its abstract syntax tree, $S(f)$ its preorder sequence of internal nodes (functions and operators), and $L(f)$ its preorder sequence of leaf nodes (references and values).

Determinism. The mapping τ is deterministic because every step in the procedure is rule-based. We first compare the skeletons $S(f^w)$ and $S(f^*)$ by length and then by the first mismatched internal-node position. Only when the two skeletons are identical do we compare leaf nodes in $L(f^w)$ and $L(f^*)$. These comparisons involve no stochastic choice, execution feedback, or manual annotation. Therefore, the same input pair (f^w, f^*) always yields the same output label.

Mutual exclusivity. For any fixed pair (f^w, f^*), the skeleton-first procedure returns at most one label. If the two skeletons differ in length, the error is assigned to either a Missing or Redundant category; if their lengths are equal but the first differing internal node has a different label, the error is assigned to a Mismatch category. These cases are disjoint. When the two skeletons are identical, we move to leaf comparison and assign either REFMISMATCH or VALMISMATCH based on the first differing leaf type. Again, these branches are disjoint. Finally, all residual or unparseable cases are assigned to MISCELLANEOUS. Hence no pair can receive two labels simultaneously.

Exhaustiveness. Every evaluated prediction is assigned some label. If f^w is unparseable, we assign MISCELLANEOUS by definition. Otherwise, both formulas are parseable and can be compared through their skeleton and leaf sequences. If the skeletons differ, the first structural discrepancy determines one of the six skeleton-level labels. If the skeletons match exactly but the leaves differ, the first leaf discrepancy determines either REFMISMATCH or VALMISMATCH. Any remaining residual case is assigned to MISCELLANEOUS. Therefore, every non-identical prediction–gold pair falls into exactly one taxonomy label.

Remark. These properties rely on two design choices of our taxonomy: (1) preorder traversal, which fixes a unique left-to-right comparison order, and (2) the skeleton-first rule, which gives structural discrepancies priority over leaf-level substitutions. Together, they ensure that the taxonomy remains simple, interpretable, and automatically derivable from AST differences.