# FormulaSPIN: Self-Play Fine-Tuning for Natural Language to Spreadsheet Formula Generation

**Cy Xie**
Cupertino, CA
ctrom0604@gmail.com

## Abstract

Spreadsheet applications are used by hundreds of millions worldwide, yet writing formulas remains a significant barrier. Existing approaches rely on static supervised data, which quickly saturates on limited annotations. In this paper, we introduce FORMULASPIN, a self-play framework that breaks the ceiling of supervised fine-tuning by enabling iterative self-improvement without any additional data. Vanilla SPIN fails on this task: it uniformly penalizes every non-matching output, so execution-equivalent alternatives are pushed down as negatives in one example while serving as ground truth in another, producing contradictory gradients. Our framework resolves this by exploiting formula generation's unique advantage: binary executability provides implicit supervision that separates semantic errors from valid stylistic variants. We frame training as a two-player game in which the main player learns to prefer ground-truth formulas over those from its previous version, while execution feedback sorts outputs into distinct granularities—enabling an adaptive curriculum that shifts from semantic correctness to stylistic refinement. To further increase accuracy, we incorporate ExecVote, a semantic-level voting mechanism that naturally handles multiple valid formulations. Experiments on multiple benchmarks demonstrate that FORMULASPIN achieves state-of-the-art performance, with 74.9% exact match and 87.1% execution accuracy on NL2FORMULA, matching models trained with additional preference annotations while outperforming both traditional SFT and frontier proprietary models. These findings underscore self-play's potential to tackle scarce data tasks and open the door to extending it beyond executable domains.

Code Implementation is available at https://github.com/xTronzZ/FormulaSPIN.

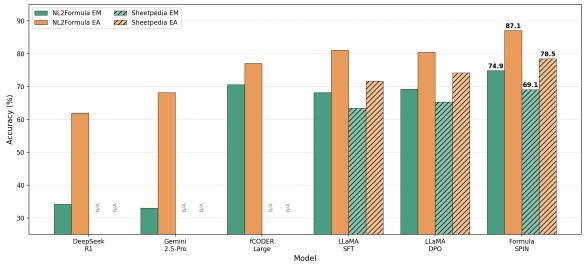


Figure 1: Performance comparison on NL2Formula-70K (Zhao et al., 2024) and Sheetpedia (Tian et al., 2025). FormulaSPIN achieves state-of-the-art results on both benchmarks without additional annotations, outperforming the SFT baseline by +6.7% EM (Exact Match) / +6.0% EA (Execution Accuracy) on NL2Formula-70K and +5.7% EM / +6.8% EA on Sheetpedia, while matching SFT-DPO trained with GPT-4.1-labeled preference pairs as well as strong proprietary models such as Gemini-2.5-Pro and DeepSeek-R1.

## 1 Introduction

Spreadsheet applications like Microsoft Excel and Google Sheets are ubiquitous tools for data analysis, used by hundreds of millions worldwide. However, writing formulas remains a significant barrier for many users, particularly when dealing with complex operations involving multiple functions, conditional logic, and cell references (Gulwani, 2011). Recent work addresses this via Natural Language to Formula (NL2Formula) generation—first systematically benchmarked by Zhao et al. (2024)—where users express their intent in plain language and the system automatically produces executable formulas.

Despite promising progress, most NL2Formula systems rely on supervised fine-tuning on static datasets, framing formula generation as a conventional seq2seq problem. However, existing datasets are limited in scale and coverage: the largest benchmark contains only 70K examples spanning limited formula patterns and table structures. Although recent work Sheetpedia (Tian et al., 2025) pro-



*Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14457–14471

vides over 290K worksheets, it serves as a general-purpose resource; fine-tuning on it yields only 2.1% improvement over the 70K benchmark despite the much larger corpus. This indicates that simply scaling static data offers limited benefit, as training on fixed datasets quickly plateaus, with additional epochs producing diminishing returns or even degrading performance (Chen et al., 2024b). Static supervision offers no clear path forward: collecting preference data for RLHF or RLAIF reintroduces the same bottleneck, requiring extensive manual evaluation or access to proprietary models. Current systems thus remain far below their potential, particularly struggling with complex nested formulas and ambiguous queries where supervision is scarcest. Overcoming this ceiling requires a fundamentally different paradigm, enabling models to improve beyond initial supervision without additional annotations or external preference signals.

To address these challenges, we introduce FormulaSPIN, a self-play fine-tuning framework specifically designed for spreadsheet formula generation. Drawing inspiration from AlphaGo Zero's self-play mechanism (Silver et al., 2017) and recent advances in self-improving language models (Chen et al., 2024b), our approach enables iterative model improvement through a two-player game: the current model (main player) learns to distinguish formulas generated by its previous version (opponent) from ground-truth formulas, while the opponent strives to generate formulas indistinguishable from human-written ones.

The key insight underlying FormulaSPIN is that formula generation offers a unique advantage over general generation tasks: executability provides implicit supervision. Unlike open-ended text where quality assessment requires human judgment, formulas can be automatically validated by executing them in a spreadsheet engine. When compared to code generation that also incorporates execution-based supervision, formula execution is *binary, deterministic, and lightweight*, requiring no manually designed test cases or complex runtime environments. Critically, vanilla self-play fails for this task because it treats all non-matching formulas uniformly, penalizing execution-equivalent alternatives (e.g., `SUM(A1:A5)` vs. `A1+A2+A3+A4+A5`) and creating contradictory training signals.

We address this through three novel components:

- **Formula-Aware Self Play**: incorporates execution feedback into the self-play objective, categorizing generated formulas by error type to weight their contribution appropriately.
- **Multi-Granularity Curriculum**: automatically adjusts training focus based on the distribution of semantic errors versus stylistic variations, progressing from correctness to style as the model matures.
- **Execution-based Voting**: generates multiple candidates at inference time and selects the best one via execution-based semantic voting.

Our contributions are fourfold. **(1)** We identify a fundamental limitation of supervised learning for NL2Formula and reformulate spreadsheet formula generation as an executable self-improvement problem, where static annotations alone are insufficient for learning complex formulas and ambiguous queries. **(2)** We propose FormulaSPIN, a self-play fine-tuning framework that leverages the binary executability of formulas to provide intrinsic supervision, enabling iterative improvement without additional human preferences or external teacher models. **(3)** We introduce an adaptive semantic-to-stylistic curriculum that automatically shifts training focus from correctness to canonical formulation as the model improves. **(4)** Through extensive in-domain and out-of-domain experiments, we demonstrate that FormulaSPIN substantially outperforms supervised fine-tuning and matches preference-optimized models trained with large-scale additional annotations, with particularly strong gains on complex nested formulas.

## 2 Related Work

Our work addresses three fundamental challenges in formula generation: *(i)* limited training data that constrains model performance, *(ii)* difficulty in obtaining quality supervision signals, and *(iii)* handling multiple valid solutions at inference time. We organize related work around how prior approaches tackle these challenges and where they fall short.

### 2.1 Formula Generation and Data Bottleneck

Semantic parsing evolved from logic forms (Price, 1990; Zelle and Mooney, 1996) through knowledge-base grounding (Berant and Liang, 2014) to SQL-based systems (Zhong et al., 2017; Yu et al., 2018). For formula synthesis, FlashFill (Gulwani, 2011) pioneered programming-by-example. Structure-aware pre-training methods model table layouts

(Wang et al., 2021; Liu et al., 2022) and formula semantics (Cheng et al., 2022). Recently, Zhao et al. (2024) introduced NL2Formula with 70K examples, and Tian et al. (2025) constructed Sheetpedia with 290K samples.

These approaches face inevitable saturation on fixed datasets, particularly for complex nested formulas where supervision is scarcest. Our work breaks this ceiling through iterative self-improvement, generating training signal from the model's own outputs without additional annotation.

### 2.2 From Supervision to Self-Improvement

Given the data bottleneck, recent work explores alternative supervision: external preference collection versus self-generated feedback.

Preference-based methods (Ouyang et al., 2022; Bai et al., 2022; Rafailov et al., 2023) require expensive human annotations or proprietary teacher models, shifting the bottleneck. For formula generation, distilling GPT-4.1 introduces external dependencies and annotation costs.

Self-play mechanisms, from TD-Gammon (Tesauro, 1995) to AlphaGo (Silver et al., 2017), enable improvement by competing against oneself. SPIN (Chen et al., 2024b) adapted this to language models, but vanilla SPIN fails for formula generation: it treats non-matching outputs uniformly, penalizing execution-equivalent alternatives (e.g., `SUM(A1:A5)` vs. `A1+A2+A3+A4+A5`) and creating contradictory gradients.

We introduce formula-aware self-play that exploits execution feedback to distinguish semantic errors from stylistic variations, enabling stable learning without external supervision.

### 2.3 Execution-Based Learning for Formulas

Execution-based validation has proven effective in code generation (Le et al., 2022; Chen et al., 2024a), but code execution requires test suites and runtime environments. For spreadsheet formulas, recent work has explored domain-specific reinforcement learning with execution feedback. Fortune (Cao et al., 2025) applies PPO with symbol-level rewards that encourage function-argument compatibility and penalize syntax errors. However, such fine-grained reward shaping demands extensive domain knowledge and remains sensitive to reward design choices. Prior spreadsheet work either ignores executability or uses it only for post-hoc filtering (Cheng et al., 2022; Yang et al., 2022).

Formula execution is binary, deterministic, and lightweight—a single table provides complete validation. We exploit this to filter and categorize formulas into semantic versus stylistic errors, constructing an adaptive curriculum. Unlike Fortune's symbol-level rewards or general RL methods (Shojaee et al., 2023; Shinn et al., 2024) that use sparse binary rewards, our preference-based objective provides richer learning signals by contrasting model outputs with ground-truth references. This also contrasts with constrained decoding (Scholak et al., 2021) that guarantees syntax but not semantics.

### 2.4 Test-Time Scaling by Semantic Consensus

Inference-time computation improves generation quality (Snell et al., 2024), but existing strategies have limitations. Best-of-N sampling (Cobbe et al., 2021) requires trained verifiers; self-consistency (Wang et al., 2023) votes on surface forms, missing semantically equivalent solutions. For code, sampling multiple solutions boosts performance (Chen et al., 2021), but verification demands comprehensive test suites.

We introduce ExecVote (Execution-based Voting) that votes over execution results rather than surface forms. By grouping formulas into semantic equivalence classes through execution, we naturally handle multiple valid formulations without trained verifiers. Among execution-equivalent candidates, we select the highest-probability formula, combining semantic correctness with stylistic preference.

## 3 FormulaSPIN

### 3.1 Problem Formulation

Given a spreadsheet table $\mathcal{T}$ and a natural language query $q$, the NL2Formula task aims to generate an executable formula $f$ that fulfills the user intent. We denote the training dataset as $\mathcal{D} = \{(q_i, \mathcal{T}_i, f_i)\}_{i=1}^{N}$, where the model learns a conditional distribution $p_\theta(f \mid q, \mathcal{T})$. Each table is serialized in row-major order with cell coordinates (e.g., "A1:Revenue | B1:2023 | ...").

### 3.2 Formula-Aware Self Play

We adopt a self-play fine-tuning framework inspired by SPIN (Chen et al., 2024b), tailored to spreadsheet formula generation. At iteration $t$, the old model $p_{\theta_t}$ acts as opponent player and generates candidate formulas $f'$ for a given query $q$ and table $\mathcal{T}$, while new model $p_{\theta_{t+1}}$ serves as the main player and is trained to assign higher probability to

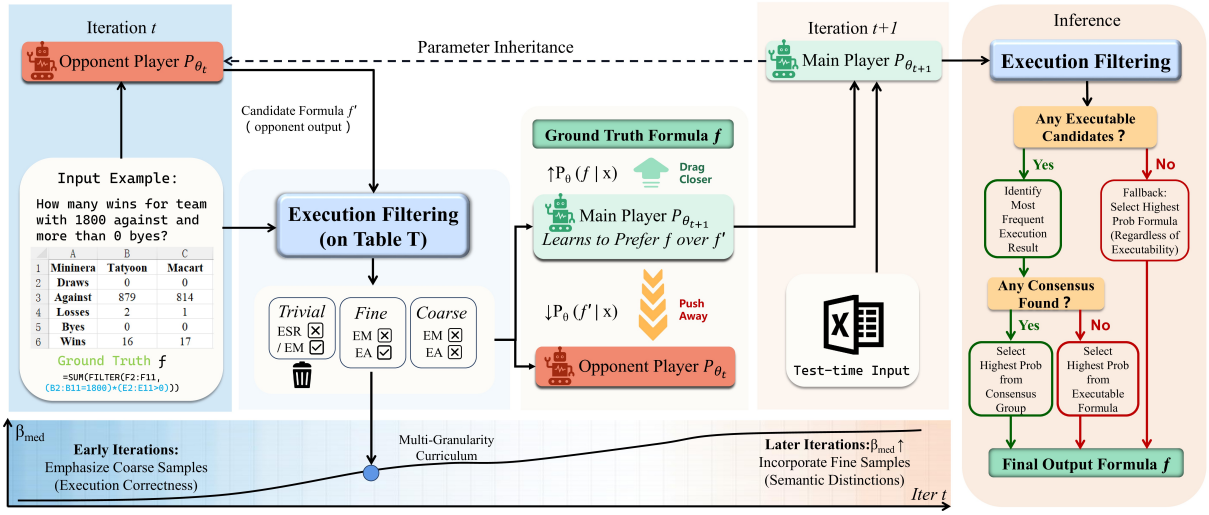


Figure 2: **The FormulaSPIN framework.** The model iteratively self-improves via a two-player game, utilizing Execution Filtering to construct an adaptive curriculum that progresses from semantic correctness (*Coarse* samples) to stylistic refinement (*Fine* samples). At inference, an execution-based voting strategy selects the final formula based on execution agreement among sampled candidates.

reference formula $f$ than to $f'$. As self-play progresses, the opponent produces increasingly challenging formulas, pushing the main player toward finer-grained semantic understanding.

A key advantage of formula generation is deterministic executability, which we leverage through *execution filtering*: comparing $\mathcal{E}(f', \mathcal{T})$ with $\mathcal{E}(f, \mathcal{T})$ sorts each candidate into *Trivial*, *Coarse*, or *Fine* samples (Figure 4). Execution failures are folded into Trivial because their diverse error patterns provide inconsistent gradients, and the SFT model already achieves ∼96% execution success. The Coarse/Fine split lets us disentangle semantic errors from stylistic variations and weight them differently in the loss.

Building on this filtering mechanism, we measure the relative preference between models using the log-probability difference

$$\mathcal{L}(\theta; \theta_t) = \mathbb{E}\left[w(f') \cdot \ell\left(\lambda \log \frac{p_\theta(f \mid q, \mathcal{T})}{p_{\theta_t}(f \mid q, \mathcal{T})} - \lambda \log \frac{p_\theta(f' \mid q, \mathcal{T})}{p_{\theta_t}(f' \mid q, \mathcal{T})}\right)\right], \tag{1}$$

where the expectation is computed over the distribution $q \sim \mathcal{Q}(\cdot)$, $f \sim p_{\text{data}}(\cdot \mid q, \mathcal{T})$, $f' \sim p_{\theta_t}(\cdot \mid q, \mathcal{T})$, and $\ell(t) := \log\big(1 + \exp(-t)\big)$ denotes the logistic loss.

The relative form $r(f) - r(f')$ encodes the adversarial dynamic: the main player must outperform the opponent's ability to distinguish reference from synthetic formulas, and as the opponent strengthens, the margin for improvement narrows, compelling progressively finer discrimination. The weight $w(f')$ is determined by execution filtering:

$$w(f') = \begin{cases} 1, & \text{if } f' \text{ is } \textit{Coarse}, \\ \beta_{\text{med}}, & \text{if } f' \text{ is } \textit{Fine}, \\ 0, & \text{if } f' \text{ is } \textit{Trivial}. \end{cases} \tag{2}$$

i.e., Coarse formulas receive full weight, Trivial formulas receive zero weight, and Fine formulas receive an adjustable weight $\beta_{\text{med}} \in [0, 0.25]$.

The choice of $\beta_{\text{med}}$ for Fine samples addresses a key tension: While Fine samples are execution-equivalent to the reference and thus semantically correct, the self-play objective still treats them as negatives to be pushed away. Assigning them the same weight as Coarse samples would create contradictory gradients—the same canonical form may appear as ground truth in one example and as a penalized opponent output in another (see Figure 3)—destabilizing training. Yet setting their weight to zero is also undesirable: reference formulas are typically more concise and idiomatic than execution-equivalent alternatives (e.g., `SUM(A1:A5)` vs. `A1+A2+A3+A4+A5`), and we want the model's outputs to converge toward these canonical forms rather than drift into verbose variants.

We resolve this tension by down-weighting Fine samples with $\beta_{\text{med}} \in [0, 0.25]$, and letting $\beta_{\text{med}}$ grow with the Fine/Coarse ratio: early on, when Coarse dominates, we avoid penalizing correct alternatives almost entirely; later, once semantic errors are rare, a mild penalty on non-canonical forms nudges the model toward idiomatic style without

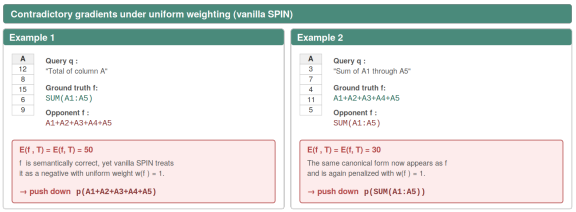


Figure 3: Contradictory gradients under uniform weighting. When `SUM(A1:A5)` and `A1+A2+A3+A4+A5` swap roles across examples, vanilla SPIN pushes down both correct forms, destabilizing training.

conflicting with the correctness objective. We formalize this as:

$$\beta_{\text{med}}^{(t)} = \beta_{\max} \cdot \frac{|\mathcal{S}_{\text{fine}}^{(t)}|}{|\mathcal{S}_{\text{fine}}^{(t)}| + |\mathcal{S}_{\text{coarse}}^{(t)}|}, \tag{3}$$

where $\beta_{\max} = 0.25$ caps the maximum weight. This implements a natural curriculum—mastering correctness before refining style—without manual schedule design. While curriculum learning has been applied to code generation (Naïr et al., 2024), our approach uniquely derives the curriculum from execution feedback rather than predefined difficulty metrics.

Following the theoretical analysis in SPIN (Chen et al., 2024b), our execution-aware filtering effectively redefines $p_{\text{data}}$ as the distribution over filtered, execution-validated samples. By excluding only execution-failing negatives while preserving semantically equivalent alternatives, the SPIN convergence guarantee remains intact: the global optimum of $\mathcal{L}$ is achieved if and only if $p_\theta = p_{\text{data}}$ under this redefined target, ensuring that iterative self-play drives the model toward the target distribution. In practice, the opponent generates increasingly sophisticated formulas across iterations—from simple errors and misuses early on to subtle semantic mistakes later—forcing the main player to develop progressively finer-grained discrimination capabilities. Figure 6 visualizes this convergence, showing diminishing marginal gains beyond iteration 3.

### 3.3 Execution-based Voting

Unlike best-of-$N$ sampling (Nakano et al., 2021; Cobbe et al., 2021), which requires external verifiers and additional supervision, *ExecVote* leverages the deterministic constraints of formula execution as a training-free validity filter. While self-consistency methods (Wang et al., 2023) count surface-level matches, we group candidates based on their semantic behavior during execution. By doing so, we capture the collective strength of semantically equivalent programs (Chen et al., 2021) that would otherwise be treated as distinct incorrect samples.

Specifically, we sample $K$ candidate formulas at temperature $T > 1$. Non-executable samples are discarded, and voting is conducted on the resulting execution outputs. We select the highest-probability formula from the majority result set; if no consensus exists, the most probable executable formula is returned, falling back to the overall top-1 candidate if all executions fail.

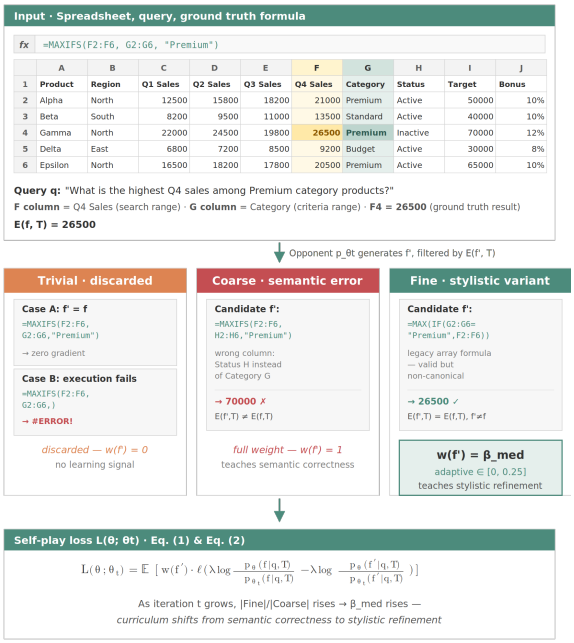


Figure 4: **Execution filtering and the Formula-Aware Self-Play objective.** The opponent $p_{\theta_t}$ generates $f'$, which is sorted by executing against $\mathcal{T}$: *Trivial* samples (exact matches or execution failures) are discarded; *Coarse* samples (wrong result) receive full weight to teach semantic correctness; *Fine* samples (correct result, non-canonical form) receive adaptive weight $\beta_{\text{med}}$ for stylistic refinement.

## 4 Experiments

### 4.1 Experimental Setup

**Benchmarks.** We evaluate on two benchmarks: NL2Formula-70K (Zhao et al., 2024), which contains 70,799 query–formula pairs from 21,670 tables covering 37 function types and uses the official train/validation/test split of 75%/10%/15%, and Sheetpedia-Selected(Tian et al., 2025), which consists of 2,167 high-quality, human-verified examples drawn from a 290K spreadsheet corpus.

**Algorithm 1** FormulaSPIN Training

**Require:** Dataset $D$, model $\theta_0$, iterations $K$
**Ensure:** Trained model $\theta_K$

**for** $t = 0$ to $K - 1$ **do**
  $S \leftarrow \emptyset$
  **for** each $(q, T, f) \in D$ **do**
    $f' \sim p_{\theta_t}(\cdot \mid q, T)$
    $r, r' \leftarrow \mathcal{E}(f, T), \mathcal{E}(f', T)$
    **skip if** $r'$=ERR or $f'$=$f$
    $c \leftarrow$ FINE if $r'$=$r$ else COARSE
    $S \leftarrow S \cup \{(q, T, f, f', c)\}$
  **end for**
  $n_f \leftarrow |\{s \in S : c\text{=FINE}\}|$
  $w_m \leftarrow \beta_{\max} n_f / |S|$
  **for** each batch $B \subseteq S$ **do**
    Compute $L$ via Eq. (1)
    Update $\theta$ by gradient descent
  **end for**
  $\theta_{t+1} \leftarrow \theta$
**end for**
**return** $\theta_K$

A detailed statistics analysis can be found in Appendix A.

**Training Configuration.** All models are initialized from LLaMA-3.1-8B-Instruct (Dubey et al., 2024). We first obtain an SFT base model on the NL2Formula training set using LoRA ($r = 16$, $\alpha = 16$, dropout $= 0$) and AdamW-8bit, with learning rate $3 \times 10^{-4}$, weight decay $10^{-3}$, cosine scheduling, and warmup ratio 0.03. Starting from this SFT-initialized adapter, we perform SPIN self-play fine-tuning for four iterations. In multiple iterations, we leverage the synthetic data from the most recent iteration and add to the newly generated synthetic data, therefore resulting in a synthetic dataset size of 50k at iteration 0 and 100k at iteration 1, 2 and 3. In each SPIN iteration, we train for 2 epochs with per-device batch size 1, gradient accumulation 16, BF16, RMSProp, zero weight decay, and warmup ratio 0.1. The peak learning rate is $5 \times 10^{-7}$ for iterations 0–2 and $1 \times 10^{-7}$ for iteration 3. The SPIN logit scaling coefficient is set to 0.1 for iterations 0–2 and 5.0 for the final iteration. Formula execution is conducted using a Linux-based customized spreadsheet formula execution simulator.

**Baselines.** We compare against several representative approaches, including FORTAP (Cheng et al., 2022), which builds on TUTA (Wang et al., 2021) and extends table pre-training to include spreadsheet formulas, using a two-stage LSTM decoder (Hochreiter and Schmidhuber, 1997); fCODER-Base and fCODER-Large (Zhao et al., 2024), which are T5-based sequence-to-sequence models trained with supervised fine-tuning; SFT, our LLaMA-3.1-8B-Instruct (Dubey et al., 2024) base model trained with standard supervised fine-tuning for two epochs; SFT (Continuous), which continues training LLaMA-3.1-8B-Instruct for an additional three epochs on the same data; SFT-DPO, which further trains the SFT model using Direct Preference Optimization (Rafailov et al., 2023) with wrong–correct pairs as preference data; and a suite of strong proprietary API models evaluated in a zero-shot setting, including GPT-4o (OpenAI, 2024), Claude-3.7-Sonnet (Anthropic, 2025), Gemini-2.5-Pro (Google DeepMind, 2025), and DeepSeek-R1 (DeepSeek-AI, 2025).

**Evaluation Metrics.** We evaluate model performance using four complementary metrics: Exact Match (EM), string-level exact correctness of formulas; Execution Accuracy (EA), semantic correctness via identical execution results; Execution Success Rate (ESR), percentage of executable formulas; and Formula Sketch Match (FSM), function-level correctness ignoring cell references. Following Zhao et al. (2024), we also report results by formula complexity: Simple (1–2 functions), Medium (3–4), and Complex (5+). Note that this complexity-based categorization is distinct from the Trivial/Fine/Coarse error granularity used during training, which reflects semantic rather than structural distinctions.

### 4.2 Main Results

**Overall Performance.** Table 1 shows results on the NL2Formula-70K test set. FormulaSPIN achieves 74.9% EM and 87.1% EA, substantially outperforming all baselines. All reported results are averaged over three runs with different random seeds. We highlight several key findings:

- Self-play provides consistent gains: +1.9% EM at iter 0, accumulating to +6.7% EM by iter 3 over the SFT base.
- FormulaSPIN (iter 3) outperforms SFT-DPO despite using no additional data. DPO uses GPT-4.1 for preference data, yet still underperforms, suggesting that execution-based intrinsic feedback is superior to distillation from external LLMs.
- Continued SFT degrades performance (-1.9%

| Model | EM | EA | ESR | FSM |
|---|---|---|---|---|
| FORTAP | 24.2 | - | - | 58.4 |
| fCODER-Large | 70.6 | 77.1 | 96.5 | 95.1 |
| SFT (base) | 68.2 | 81.1 | 97.1 | 95.7 |
| SFT (Continued) | 68.8 | 79.2 | **99.4** | 92.6 |
| SFT-DPO | 72.2 | 84.5 | 98.6 | 94.9 |
| *API Models* | | | | |
| DeepSeek-R1 | 34.2 | 62.0 | 88.6 | - |
| Gemini-2.5-Pro | 33.0 | 68.2 | 92.4 | - |
| Claude-3.7-Sonnet | 32.4 | 61.2 | 87.1 | - |
| GPT-4o | 34.0 | 66.1 | 91.3 | - |
| *Ours* | | | | |
| **FormulaSPIN** ($t_0$) | 70.1 | 82.5 | 98.8 | 96.1 |
| **FormulaSPIN** ($t_1$) | 72.5 | 84.7 | 99.1 | 97.8 |
| **FormulaSPIN** ($t_2$) | 74.1 | 86.9 | 99.1 | 98.3 |
| **FormulaSPIN** ($t_3$) | **74.9** | **87.1** | 99.2 | **98.7** |

Table 1: Performance on NL2Formula-70K test set. **EM**: Exact Match; **EA**: Execution Accuracy; **ESR**: Execution Success Rate; **FSM**: Formula Sketch Match.

EA), confirming naive multi-epoch training ineffective.

- Even large proprietary models underperform our approach, showing the value of task-specific self-play fine-tuning.

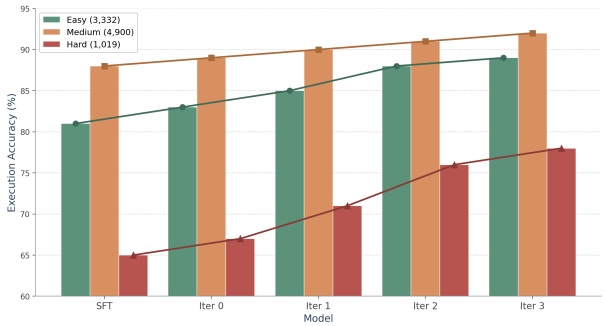


Figure 5: Execution Accuracy by formula complexity.

**Performance by Complexity.** Figure 5 analyzes results across formula complexity levels. The largest improvements occur on complex formulas (+11.0% from SFT to iter 3), with the bulk of the gain concentrated in iterations 1 and 2 (+4.0 and +5.0 points, respectively) before plateauing at iter 3—suggesting self-play effectively discovers compositional patterns that SFT alone fails to capture. Simple formulas also benefit notably (+8.0%), while medium formulas, which already attain the highest absolute accuracy under SFT, exhibit the smallest headroom and gain a more modest +4.0%. This pattern indicates that self-play is most beneficial precisely where SFT struggles the most, narrowing the gap between difficulty tiers as iterations progress.

### 4.3 Generalization and Scaling

**Execution-based Voting.** Table 2 shows the effect of sampling multiple candidates at inference. Sampling $K = 10$ candidates boosts EA by +7.0% over greedy decoding, at roughly 1.68× the inference cost. The gains saturate beyond $K = 10$, suggesting diminishing returns. This demonstrates that formula generation benefits significantly from test-time compute, similar to code generation (Chen et al., 2021).

| K (samples) | EA | Inference Time |
|---|---|---|
| 1 (greedy) | 86.7 | 1.00× |
| 5 | 91.4 | 1.32× |
| 10 | 93.8 | 1.68× |

Table 2: Impact of test-time compute scaling (FormulaSPIN iter 3, 128 test samples). Sampling $K = 10$ provides the best accuracy-efficiency tradeoff.

**Out-of-Domain Generalization.** We evaluate on the Sheetpedia-NL2FL test set (2167 examples from a different data distribution) to assess generalization. The SFT baseline achieves 63.4% EM and 71.7% EA, while SFT-DPO improves to 65.3% EM and 74.2% EA. FormulaSPIN generalizes well to out-of-domain data, reaching 69.1% EM and 78.5% EA. Combined with test-time scaling ($K = 10$), performance further increases to 84.9% EA.

**Robustness Across Base Models.** To verify that these improvements generalize across model architectures, we also evaluate FormulaSPIN on multiple base models including Qwen3-8B, Qwen2.5-7B, DeepSeek-Coder-7B, and Mistral-7B, observing consistent gains of +6–8% EM across all architectures.

### 4.4 Training Dynamics

**Iteration Analysis.** Figure 6 tracks training dynamics across iterations. On NL2Formula, improvements are most pronounced in the early iterations (+2.4% EM from iter 0 to 1, +1.6% from iter 1 to 2) and gradually diminish, with EM rising from 70.1% at iter 0 to 74.9% at iter 3 before plateauing. Sheetpedia (out-of-domain) exhibits a similar trajectory, climbing from 63.4% to 69.1% EM over the first four iterations and then showing only minor fluctuations around 69% (69.1% → 68.9% → 69.2%). Execution Accuracy follows the same pattern on both benchmarks, converging near 87.1% on NL2Formula and 78.5% on Sheetpedia.

This convergence aligns with theoretical predictions (Chen et al., 2024b) and indicates that 3–4 iterations provide the optimal trade-off between performance and training cost.

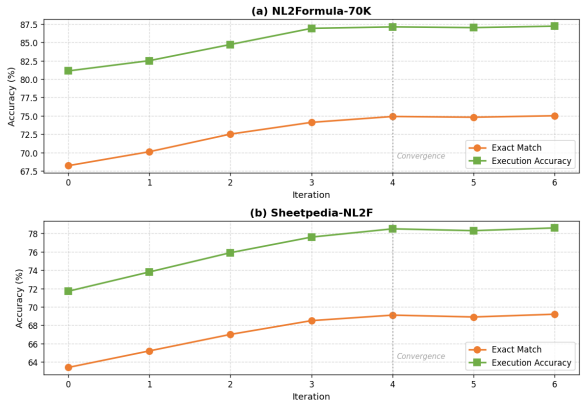


Figure 6: Performance progression across iterations on (a) NL2Formula-70K and (b) Sheetpedia-NL2FL.

| Iteration | Exact Match % | Execution Match % | Syntax Error % |
|---|---|---|---|
| 0 (from SFT) | 71.4 | 83.9 | 4.6 |
| 1 | 73.8 | 85.6 | 3.3 |
| 2 | 75.2 | 87.4 | 2.5 |
| 3 | 76.1 | 88.3 | 1.8 |

Table 3: Quality of self-generated training formulas across iterations. As the model improves, synthetic data quality increases, creating a virtuous cycle.

**Synthetic Data Quality.** We analyze the quality of self-generated formulas used in training (Table 3). Synthetic data quality improves steadily across iterations: exact matches rise from 71.4% to 76.1% (+4.7%), execution matches climb from 83.9% to 88.3% (+4.4%), and syntax errors drop from 4.6% to 1.8%. Notably, synthetic data quality is consistently slightly higher than test-set accuracy as our generation pipeline applies confidence-based filtering and rejection sampling to discard low-quality candidates. This virtuous cycle—a stronger model produces higher-quality training data, which in turn trains an even stronger model—is central to the effectiveness of self-play, and it also explains the diminishing gains observed beyond iteration 3–4: as synthetic data quality approaches the model's own ceiling, the additional learning signal naturally saturates.

### 4.5 Ablation Studies

**Components Analysis.** Table 4 ablates the key components of FormulaSPIN. Disabling the adaptive curriculum and reverting to a fixed $\beta_{\text{med}}$ leads to a -1.9% EM and -1.5% EA degradation, validating that progressively shifting focus from semantic correctness to stylistic refinement is more effective than a static weighting scheme.

| Configuration | EM | EA |
|---|---|---|
| FormulaSPIN | **74.9** | **87.1** |
| - w/o adaptive curriculum | 73.0 | 85.6 |
| - w/ vanilla SPIN | 70.6 | 82.4 |
| - w/o self-play (SFT only) | 68.2 | 81.1 |

Table 4: Ablation study on NL2Formula-70K removing key components of FormulaSPIN. The adaptive curriculum contributes substantially, while vanilla SPIN degrades performance after the first iteration.

Most strikingly, vanilla SPIN starts *degrading* performance after iter0. This counterintuitive result arises because vanilla SPIN treats all non-matching outputs uniformly as negatives, penalizing execution-equivalent alternatives (e.g., `SUM(A1:A5)` vs. `A1+A2+A3+A4+A5`) alongside genuinely incorrect formulas. The resulting contradictory gradients destabilize training and erode the gains achievable through self-play. By contrast, FormulaSPIN's formula-aware self-play objective combined with the adaptive curriculum yields a +6.7% EM and +6.0% EA improvement over SFT, demonstrating that both components are indispensable for unlocking the benefits of self-play in formula generation. A more detailed analysis of vanilla SPIN's failure modes is provided in Appendix C.

**Adaptive Curriculum Analysis.** We validate our adaptive weighting mechanism (Equation 3) against fixed $\beta_{\text{med}}$ values. The adaptive approach outperforms both fixed weights and a hand-tuned linear schedule. The key advantage is that $\beta_{\text{med}}$ automatically tracks training progress: it starts low when Coarse samples dominate, then increases as Fine samples become prevalent. This implements a natural semantic-to-stylistic curriculum without manual tuning.

| Schedule | EM | EA |
|---|---|---|
| Fixed $\beta = 0.0$ | 70.1 | 86.9 |
| Fixed $\beta = 0.10$ | 72.3 | 85.4 |
| **Adaptive** | **74.9** | **87.1** |

Table 5: Comparison of $\beta_{\text{med}}$ scheduling strategies.

### 4.6 Qualitative Analysis

**Case Study.** Figure 7 presents a case study on conditional aggregation. The table in `A1:J6` contains the NL query "*What is the highest Q4 sales among Premium category products?*" The ground-truth formula is `MAXIFS(F2:F6, G2:G6, "Premium")`, yielding "26500". The SFT model predicts `MAX(IF(G2:G6="Premium", F2:F6))`, a legacy array formula that requires Ctrl+Shift+Enter in older Excel versions and fails silently in certain environments. While FormulaSPIN learns the modern `MAXIFS` pattern through self-play despite the sparsity of relevant data.

| | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Product | Region | Q1 Sales | Q2 Sales | Q3 Sales | Q4 Sales | Category | Status | Target | Bonus |
| 2 | Alpha | North | 12500 | 15800 | 18200 | 21000 | Premium | Active | 50000 | 10% |
| 3 | Beta | South | 8200 | 9500 | 11000 | 13500 | Standard | Active | 40000 | 10% |
| 4 | Gamma | North | 22000 | 24500 | 19800 | **26500** | **Premium** | Inactive | 70000 | 12% |
| 5 | Delta | East | 6800 | 7200 | 8500 | 9200 | Budget | Active | 30000 | 8% |
| 6 | Epsilon | North | 16500 | 18200 | 17800 | 20500 | Premium | Active | 65000 | 10% |

Figure 7: An example where SFT produces a legacy array formula requiring special entry mode, while FormulaSPIN generates the modern idiomatic function.

**Error Analysis.** We conduct stratified random sampling (300 examples each from Simple, Medium, and Complex buckets) on FormulaSPIN ($t_3$) outputs and find that both *error categories* and *function-family involvement* shift systematically with formula difficulty. In the *Simple* bucket, failures concentrate on aggregation choice and over-complication—confusions involving `SUM` as either reference or prediction account for 29.2% of errors (e.g., `SUM`→`MAX`, `SUM`→`ROWS`, `SUM`→`AVERAGE`), and the model often rewrites a direct `FILTER` as a nested `LET+SUMMARIZE` (`LET`→`FILTER`, 12.5%). In the *Medium* bucket, the model essentially stops picking the wrong family: errors remain inside `UNIQUE`→`UNIQUE` (35.3%) and `SORT`→`SORT` (23.5%) confusions and localize to *projection and value binding*—row selection is typically correct, but the model returns the wrong column (17.6%) or fails to match surface forms such as score strings, Unicode punctuation, and decimals like `.9` versus `0.9` (29.4%). In the *Complex* bucket, `LET`→`LET` confusions alone account for 63.8% of errors: the outer program skeleton is largely correct, but inner field ordering and condition bindings are misplaced (*wrong_return_column* 27.7%, *wrong_filter_value_or_threshold* 17.0%). This trajectory—from family-level mistakes and template over-complication, to within-family projection drift, to fine-grained binding drift inside correct skeletons—mirrors the semantic-to-stylistic curriculum learned during self-play: the model first masters *which* program to write, then *what* to return, and finally struggles only with *how* bindings are wired inside complex compositions. A detailed breakdown by bucket, error category, function-confusion pattern, and representative case studies appears in Appendix E.

## 5 Conclusion

In this paper, we propose a novel self-play fine-tuning framework for natural language to spreadsheet formula generation, called FormulaSPIN. FormulaSPIN leverages the intrinsic verifiability of formulas to enable iterative self-improvement without any additional costly human annotations or external teacher models. Unlike vanilla SPIN, it avoids contradictory gradients on execution-equivalent samples by *distinguishing semantic correctness from stylistic variations* through error-aware adversarial training. The proposed adaptive curriculum and semantic consensus voting show that task-intrinsic verification can effectively drive self-improvement. Our work represents the first effective application of self-play to formula generation tasks, significantly outperforming SFT, domain-specific models, and large proprietary models while matching models trained with large-scale preference annotations. These findings underscore the strong potential of combining self-play with task-intrinsic and dynamic feedback, opening promising directions for extending this execution-driven paradigm to other structured generation tasks with potentially sparse data, including SQL synthesis, code generation, and mathematical reasoning.

## Limitations

Our approach assumes access to a spreadsheet execution engine for validation, which may not be available in all deployment scenarios. The test-time compute scaling improves accuracy but increases inference latency, potentially limiting real-time applications. Additionally, self-play requires multiple training iterations , which may be prohibitive for resource-constrained settings. Finally, our evaluation focuses on English queries and Excel formulas; generalization to other languages and spreadsheet applications remains unexplored.

## Ethics Statement

This work uses publicly available benchmarks (NL2Formula-70K and Sheetpedia) containing synthetic spreadsheet examples without personally identifiable information. FormulaSPIN is designed to democratize spreadsheet usage by lowering technical barriers for non-expert users. We acknowledge that users should verify generated formulas before deployment in high-stakes scenarios, as over-reliance on automated outputs may introduce errors. Our self-play approach eliminates dependence on expensive proprietary models for preference annotation, reducing both financial and environmental costs compared to methods requiring extensive API calls.

## References


Anthropic. 2025. Claude 3.7 sonnet and claude code.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, and Cameron et al. McKinnon. 2022. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*.

Jonathan Berant and Percy Liang. 2014. Semantic parsing via paraphrasing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1415–1425.

Lang Cao, Jingxian Xu, Hanbing Liu, Jinyu Wang, Mengyu Zhou, Haoyu Dong, Shi Han, and Dongmei Zhang. 2025. FORTUNE: Formula-driven reinforcement learning for symbolic table reasoning in language models. *arXiv preprint arXiv:2505.23667*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, and Greg et al. Brockman. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024a. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. 2024b. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*.

Zhoujun Cheng, Haoyu Dong, Zhiruo Wang, Ran Jia, Jiaqi Guo, Yan Gao, Shi Han, Jian-Guang Lou, and Dongmei Zhang. 2022. FORTAP: Using formulas for numerical-reasoning-aware table pretraining. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1535–1547.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, and Reiichiro et al. Nakano. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, and Angela et al. Fan. 2024. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Google DeepMind. 2025. Gemini 2.5: Our most intelligent ai model.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 317–330.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, pages 21314–21328.

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2022. TAPEX: Table pre-training via learning a neural sql executor. In *International Conference on Learning Representations (ICLR)*.

Marwa Naïr, Kamel Yamani, Lynda Said Lhadj, and Riyadh Baghdadi. 2024. Curriculum learning for small code language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, and William et al. Saunders. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.

OpenAI. 2024. Hello GPT-4o.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, and Alex et al. Ray. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Patti J. Price. 1990. Evaluation of spoken language systems: The ATIS domain. In *Proceedings of the Workshop on Speech and Natural Language*, pages 91–95. Association for Computational Linguistics.

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

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9895–9901.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 8634–8652.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *Transactions on Machine Learning Research*.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, and Adrian et al. Bolton. 2017. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Gerald Tesauro. 1995. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68.

Zailong Tian, Zhuoheng Han, Houfeng Wang, and Lizi Liao. 2025. Sheetpedia: A 300k-spreadsheet corpus for spreadsheet intelligence and llm fine-tuning. In *Advances in Neural Information Processing Systems*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*.

Zhiruo Wang, Haoyu Dong, Ran Jia, Jia Li, Zhiyi Fu, Shi Han, and Dongmei Zhang. 2021. TUTA: Tree-based transformers for generally structured table pre-training. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 1780–1790.

Jingfeng Yang, Aditya Gupta, Shyam Upadhyay, Luheng He, Rahul Gao, and Wen-tau Hwang. 2022. Tableformer: Robust transformer modeling for table-text encoding. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 528–537.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 3911–3921.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 1050–1055.

Yuhao Zhang, Shaoming Duan, Jinhang Su, Chuanyi Liu, and Peiyi Han. 2025. SPFT-SQL: Enhancing large language model for text-to-sql parsing by self-play fine-tuning. *arXiv preprint arXiv:2509.03937*.

Wei Zhao, Zhitao Hou, Siyuan Wu, Yan Gao, Haoyu Dong, Shi Han, and Dongmei Zhang. 2024. NL2Formula: Generating spreadsheet formulas from natural language queries. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 1030–1046.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

## A Dataset Statistics

| Statistic | NL2F-70K | Sheetpedia (OOD) |
|---|---|---|
| Total examples | 70,799 | 2,167 |
| Unique tables | 21,670 | 1,847 |
| Function types | 37 | 42 |
| Avg. query len. | 11.2 | 15.7 |
| Avg. formula len. | 10.2 | 15.3 |
| Avg. table rows | 10.8 | 14.2 |
| Avg. table columns | 6.0 | 9.3 |
| *Complexity Distribution* | | |
| Simple (1–2 funcs) | 29.1% | |
| Medium (3–4 funcs) | 58.3% | |
| Complex (5+ funcs) | 12.6% | |

Table 6: Dataset statistics for NL2Formula-70K and Sheetpedia-NL2F benchmarks.

**Dataset Differences.** NL2Formula-70K and Sheetpedia differ fundamentally in their construction and content characteristics. NL2F-70K is derived by converting text-to-SQL examples (WikiSQL,

Spider) into spreadsheet contexts, which limits its coverage to SQL-compatible patterns. In contrast, Sheetpedia is curated from real-world sources including enterprise email archives (Enron) and user-contributed content from ExcelForum, capturing authentic spreadsheet usage across diverse domains such as financial modeling and data analysis. Consequently, Sheetpedia exhibits native Excel formula patterns with prevalent error-handling constructs (e.g., IFERROR wrapping VLOOKUP), nested conditional logic, and complex function co-occurrences that are rarely observed in SQL-converted datasets. This makes Sheetpedia a challenging out-of-distribution (OOD) benchmark for evaluating model generalization to real-world spreadsheet scenarios.

## B DPO Baseline Construction

**Dataset Quality Analysis.** The NL2Formula-70K dataset (Zhao et al., 2024) was collected from web sources, where the ground-truth formulas represent one possible solution rather than the canonical or most efficient implementation. Through careful manual inspection of 500 randomly sampled examples, we observed that approximately 18.3% of ground-truth formulas could be expressed more concisely or idiomatically.

For instance, consider the following examples where SFT-generated formulas are arguably superior:

| Ground Truth | SFT Output (More Efficient) |
|---|---|
| `A1+A2+A3+A4+A5` | `SUM(A1:A5)` |
| `SUMPRODUCT((B:B ="X")*(C:C))` | `SUMIF(B:B,"X",C:C)` |
| `INDEX(A:A, MATCH(MAX(B:B), B:B,0))` | `XLOOKUP(MAX(B:B), B:B,A:A)` |

Table 7: Examples where alternative formulas may be more efficient or idiomatic than ground-truth annotations.

**Motivation for Two-Stage Annotation.** Given this observation, naively setting the ground-truth formula as always preferred in DPO training could lead to suboptimal learning dynamics. The model might learn to memorize specific formula patterns from the dataset rather than developing genuine preferences for efficient, correct formulas.

**Two-Stage Preference Annotation Strategy.** To construct a stronger and more principled DPO baseline, we adopt a two-stage preference annotation strategy:

**Stage 1: Execution Validation.** For each training example $(q, \mathcal{T}, f_{gt})$, we first generate a candidate formula $f_{sft}$ using the SFT model. Both $f_{gt}$ and $f_{sft}$ are executed on the table $\mathcal{T}$. If $\mathcal{E}(f_{sft}, \mathcal{T}) \neq \mathcal{E}(f_{gt}, \mathcal{T})$ (execution results differ), we set the ground-truth as preferred: $(f_w, f_l) = (f_{gt}, f_{sft})$. If $\mathcal{E}(f_{sft}, \mathcal{T}) = \mathcal{E}(f_{gt}, \mathcal{T})$ (execution results match), we proceed to Stage 2.

**Stage 2: GPT-4.1 Preference Labeling.** When both formulas produce identical execution results, we employ GPT-4.1 to determine which formula is preferred based on criteria including conciseness, readability, idiomatic usage, and computational efficiency.

**Strategy Benefits.** This strategy ensures that semantically incorrect formulas are never favored over correct ones, that among multiple valid solutions the more efficient or idiomatic variant is preferred, and that the DPO baseline learns genuinely meaningful preferences rather than relying on arbitrary memorization.

**Resulting Dataset Statistics.** The resulting preference pairs consist of approximately 71.2% cases where ground-truth was preferred (due to execution mismatch or GPT-4.1 preference) and 28.8% cases where the SFT-generated formula was deemed superior.

## C Why Vanilla SPIN Fails for Formula Generation

We provide detailed analysis explaining why vanilla SPIN (Chen et al., 2024b) is unsuitable for formula generation, resulting in performance degradation compared to FormulaSPIN.

**Iteration-wise Performance.** Table 8 tracks vanilla SPIN across iterations. Unlike FormulaSPIN which shows consistent improvement, vanilla SPIN peaks at iteration 1 with marginal gains, then *degrades* in subsequent iterations.

| Method | Iter 0 | Iter 1 | Iter 2 | Iter 3 |
|---|---|---|---|---|
| Vanilla SPIN | 68.5 | 71.9 | 71.1 | 70.6 |
| FormulaSPIN | 68.5 | 72.5 | 74.1 | 74.9 |

Table 8: Exact Match (%) across iterations on NL2Formula-70K. Vanilla SPIN shows early saturation and subsequent degradation, while FormulaSPIN improves consistently.

**Core Issues with Vanilla SPIN.** We identify three fundamental problems that cause vanilla SPIN to fail for formula generation tasks.

- Problem 1: Penalizing Execution-Equivalent Formulas. Vanilla SPIN treats all $f' \neq f$ as negative samples. However, in formula generation, many alternatives are semantically correct.In our training data, approximately 15-20% of generated formulas at each iteration are execution-equivalent to references. Vanilla SPIN incorrectly pushes the model *away* from these valid solutions, creating contradictory learning signals.
- Problem 2: Noise from Syntax Errors. Vanilla SPIN includes all generated samples regardless of executability. Early iterations produce 10-12% syntax errors with highly diverse patterns (missing parentheses, invalid function names, malformed references). These samples provide inconsistent gradients that destabilize training rather than informing it.
- Problem 3: Conflating Semantics and Style. Without distinguishing Coarse (wrong result) from Fine (correct result, different form) samples, vanilla SPIN simultaneously optimizes for: (1) Semantic correctness: preferring formulas that compute the right value; (2) Stylistic conformity: preferring formulas that match reference syntax. These objectives can conflict—a verbose but correct formula should not receive the same penalty as an incorrect one. The uniformity prevents the model from establishing a clear learning hierarchy.

| Reference | Generated (Equivalent) |
|---|---|
| `SUM(A1:A5)` | `A1+A2+A3+A4+A5` |
| `AVERAGE(B:B)` | `SUM(B:B)/COUNT(B:B)` |
| `IF(A1>0,A1,0)` | `MAX(A1,0)` |

Table 9: Examples of execution-equivalent formula pairs incorrectly penalized by vanilla SPIN.

## D Generalization Across Base Models

We evaluate FormulaSPIN across multiple base models to demonstrate its generalizability. Table 10 presents complete results.

FormulaSPIN provides consistent improvements across all model families, confirming the robustness reported in Section 4.3. Code-specialized models (DeepSeek-Coder) and recent instruction-tuned models (Qwen3) tend to perform well, validating the connection between code and formula generation.

| Base Model | Params | SFT | SPIN | Δ |
|---|---|---|---|---|
| LLaMA-3.1-8B | 8B | 68.2 | 74.9 | +6.7 |
| Qwen3-8B | 8B | 69.5 | 76.3 | +6.8 |
| Qwen2.5-7B | 7B | 68.9 | 73.9 | +5.0 |
| DeepSeek-Coder-7B | 7B | 67.4 | 75.8 | +8.4 |
| Mistral-7B-v0.3 | 7B | 66.1 | 72.3 | +6.2 |

Table 10: FormulaSPIN performance (EM %) across different base models on NL2Formula-70K. Gains are consistent across architectures.

## E Detailed Error Analysis

We provide the full methodology, statistics, and qualitative case studies behind the error analysis summarized in the main text. All numbers in this section come from the same 900-example stratified random sample (300 each from Simple, Medium, and Complex buckets) drawn from FormulaSPIN ($t_3$) greedy outputs. The Calculation bucket is excluded as it is dominated by direct numeric reduction and does not interact with the compositional patterns we study here.

### E.1 Sampling and Annotation Methodology

For each of the three difficulty buckets (Simple: 1–2 functions, Medium: 3–4 functions, Complex: 5+ functions), we uniformly sample 300 examples from the FormulaSPIN ($t_3$) test outputs and manually label every error according to (i) one of the error categories defined in Table 11, and (ii) the (reference function → predicted function) pair at the top of the formula tree.

### E.2 Per-Bucket Statistics

Table 12 summarizes performance and residual-error rates per bucket. The bulk of remaining headroom lies in the Complex bucket, where execution accuracy drops by roughly 10 points relative to Simple and Medium, while Simple and Medium errors are sparse and increasingly localized.

| Bucket | EM | EA | ESR | Errors |
|---|---|---|---|---|
| Simple | 59.0 | 92.0 | 100.0 | 24 (8.0%) |
| Medium | 82.0 | 94.3 | 100.0 | 17 (5.7%) |
| Complex | 82.7 | 84.3 | 97.0 | 47 (15.7%) |

Table 12: Per-bucket statistics on the 900-example stratified sample.

| Category | Definition |
|---|---|
| *top function mismatch* | The outermost function in the predicted formula differs from the reference (e.g., `SUM` vs. `ROWS(UNIQUE(...))`). |
| *aggregation confusion* | The outer function is a reduction operator but the wrong one (e.g., `SUM` substituted for `MAX`, `AVERAGE`, or `ROWS`). |
| *lookup-strategy confusion* | A query solvable by a single `FILTER` is rewritten as a nested `LET`+`SUMMARIZE` composition, or vice versa. |
| *wrong return column* | Row selection and top-level function are correct, but the projected column index inside `CHOOSECOLS` (or equivalent) is wrong. |
| *wrong column binding* | The function selects rows or returns values from the wrong source column entirely (e.g., `A2:A6` instead of `B2:B6`). |
| *wrong filter value / threshold* | Filter structure is correct but a literal value or comparison threshold differs from the reference, including surface-form mismatches (Unicode, whitespace, decimal formatting). |
| *missing or extra filter condition* | The predicate has one fewer or one more conjunct than the reference. |
| *structural drift* | The formula tree shape diverges substantially from the reference even though both share top-level intent. |
| *same-sketch wrong argument binding* | The function tree matches the reference exactly, but argument order inside a multi-argument call (e.g., `HSTACK`) is permuted. |

Table 11: Error category taxonomy used to label the 900 stratified samples.

## E.3 Error Category Distribution

We group errors into the categories defined above. Table 13 reports their distribution per bucket; bold numbers highlight the dominant category in each column. The shape of the distribution changes qualitatively across buckets: Simple errors are spread across template-selection categories (*top function mismatch*, *aggregation confusion*, *lookup-strategy confusion*); Medium errors collapse onto *top function mismatch* and surface-form mismatches; Complex errors localize to *wrong return column* and *wrong filter value / threshold* inside otherwise correct `LET` skeletons.

| Error category | Simple | Medium | Complex |
|---|---|---|---|
| top function mismatch | **29.2** | **41.2** | 23.4 |
| aggregation confusion | 16.7 | – | – |
| lookup-strategy confusion | 16.7 | – | – |
| wrong column binding | 12.5 | 11.8 | 4.3 |
| wrong filter value | 8.3 | 29.4 | 17.0 |
| wrong return column | – | 17.6 | **27.7** |
| missing/extra filter cond. | 8.3 | – | 4.3 |
| structural drift | 4.2 | – | – |
| same-sketch arg. binding | 4.2 | – | – |

Table 13: Error category distribution (% of bucket errors) for FormulaSPIN ($t_3$). "–" indicates the category did not appear in that bucket's sample.

## E.4 Function Confusion Patterns

Table 14 reports the top (reference → predicted) function pairs per bucket. The Simple bucket is dominated by `SUM`-related confusions (29.2%) and `LET`→`FILTER` over-complication (12.5%). The Medium bucket is dominated by within-family confusions: `UNIQUE`→`UNIQUE` (35.3%) and `SORT`→`SORT` (23.5%), meaning the model picks the right family but mis-projects. The Complex bucket is dominated by `LET`→`LET` (63.8%), meaning the model picks the right top-level template but misplaces inner bindings.

| Top confusion (ref → pred) | % of bucket errors |
|---|---|
| *Simple* | |
| `SUM`-family confusions (combined) | 29.2 |
| `LET`→`FILTER` | 12.5 |
| `SUMIFS`→`ROWS` | 8.3 |
| *Medium* | |
| `UNIQUE`→`UNIQUE` | 35.3 |
| `SORT`→`SORT` | 23.5 |
| `SUMIFS`→`ROWS` | 17.6 |
| *Complex* | |
| `LET`→`LET` | 63.8 |
| `SORT`→`SORT` | 6.4 |
| `LET`→`ROWS` | 6.4 |

Table 14: Top function-confusion patterns by bucket. The transition from cross-family to within-family to within-template confusions is monotone with difficulty.

## E.5 Representative Case Studies

We illustrate each bucket's characteristic failure mode with a representative case drawn from our annotated set.

**Simple — aggregation confusion.** The model substitutes `SUM` for the correct reduction operator. Both predictions execute successfully but produce semantically wrong values.

*Query:* "What is the best top 10 when there are fewer than 0 wins?"
*Pred:* `SUM(FILTER(E2:E12, D2:D12<0))`
*Ref:* `MAX(FILTER(E1, D1<0))`

**Simple — over-complication.** A query solvable by a single `FILTER` is rewritten as a two-level `LET+SUMMARIZE` composition.

*Query:* "What is the Duration for less than 53 consecutive wins?"
*Pred:*
```
LET(query1,
SUMMARIZE(D2:D8,
SUMX(B2:B8)),
    FILTER(query1,
CHOOSECOLS(query1,2)<53))
```
*Ref:* `FILTER(D1, B1<53)`

**Medium — surface-form mismatch.** Row selection is correct, but the literal value mismatches the table due to Unicode/whitespace differences.

*Query:* "...best fit (WMAP only) is .9 ± .1..."
*Pred:* filter on `C2:C11="0.9 ± 0.1"`
*Ref:* filter on `C1=".9 ± .1"`

**Medium — wrong return column.** The model selects the right rows and the right top-level function (`UNIQUE`→`UNIQUE`) but projects the wrong column.

*Query:* "Who won the match when the winner used the Pedigree attack?"
*Pred:* `...CHOOSECOLS(FILTER(...), 2)`
*Ref:* `...CHOOSECOLS(FILTER(...), 4)`

**Complex — inner field-ordering inside `LET`→`LET`.** The outer `LET` skeleton matches the reference; the `HSTACK` argument order inside is swapped.

*Query:* "Show ids for all aircrafts with more than 1000 distance."
*Pred:*
```
LET(...,
SUMMARIZE(HSTACK(D2:D17,
A2:A17), ...), ...)
```
*Ref:*
```
LET(...,
SUMMARIZE(HSTACK(A2:A17,
D2:D17), ...), ...)
```

## F Comparison with SPFT-SQL

We compare FormulaSPIN with SPFT-SQL (Zhang et al., 2025), a concurrent work applying self-play to Text-to-SQL. Although both methods adapt SPIN to executable generation tasks, they address fundamentally different problems arising from distinct domain properties.

**The Execution-Equivalence Problem is Domain-Specific.** The core challenge in formula generation is that 15–20% of model-generated formulas are syntactically different from the ground truth yet produce identical execution results (e.g., `SUM(A1:A5)` vs. `A1+A2+A3+A4+A5`). Vanilla SPIN penalizes all non-matching outputs uniformly, causing the same canonical form to be rewarded in one example while being pushed away as a negative in another—creating systematic gradient conflicts during training. This failure mode has no analog in Text-to-SQL: syntactically distinct SQL queries rarely produce identical outputs across all database rows, so execution-equivalence is not a prevalent source of training instability.

**Solution Granularity.** FormulaSPIN addresses gradient-level conflicts by rethinking the training objective itself—introducing execution-aware sample categorization (Trivial/Coarse/Fine) with adaptive weighting to prevent contradictory gradients on valid alternatives. SPFT-SQL uses execution feedback as a static data-quality filter and reward signal, but does not confront gradient conflicts during optimization because such conflicts are rare in their domain. Their primary contribution is iterative data synthesis (VBI-FT) to combat overfitting from limited supervision—a complementary concern orthogonal to ours.