PrefGen: A Preference-Driven Methodology for Secure Yet Gas-Efficient Smart Contract Generation

Zhiyuan Peng[1†] Xin Yin[2†] Zijie Zhou[3] Chenhao Ying[1*] Chao Ni[2*] Yuan Luo[1*]
[1]*Shanghai Jiao Tong University*, {pzy2000, yingchenhao, yuanluo}@sjtu.edu.cn
[2]*The State Key Laboratory of Blockchain and Data Security, Zhejiang University*, {xyin, chaoni}@zju.edu.cn
[3]*China University of Petroleum (Beijing)*, zhouzijie@student.cup.edu.cn

***Abstract*—While Large Language Models (LLMs) have demonstrated remarkable progress in generating functionally correct Solidity code, they continue to face critical challenges in producing gas-efficient and secure code, which are critical requirements for real-world smart contract deployment. Although recent advances leverage Supervised Fine-Tuning (SFT) and Direct Preference Optimization (DPO) for code preference alignment, existing approaches treat functional correctness, gas optimization, and security as independent objectives, resulting in contracts that may achieve operational soundness but suffer from prohibitive execution costs or dangerous vulnerabilities. To address these limitations, we propose PrefGen, a novel framework that extends standard DPO beyond human preferences to incorporate quantifiable blockchain-specific metrics, enabling holistic multi-objective optimization specifically tailored for smart contract generation. Our framework introduces a comprehensive evaluation methodology with four complementary metrics: Pass@k (functional correctness), Compile@k (syntactic correctness), Gas@k (gas efficiency), and Secure@k (security assessment), providing rigorous multi-dimensional contract evaluation. Through extensive experimentation, we demonstrate that PrefGen significantly outperforms existing approaches across all critical dimensions, achieving 66.7% Pass@5, 58.9% Gas@5, and 62.5% Secure@5, while generating production-ready smart contracts that are functionally correct, cost-efficient, and secure.**

***Index Terms*—Smart Contract Generation, Supervised Fine-Tuning, Direct Preference Optimization**

I. Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in generating functionally correct Solidity code, as evidenced by recent benchmarks [1], [2]. However, while functional correctness represents a fundamental requirement, human preferences for smart contracts extend far beyond mere operational soundness. In practice, developers and users demand contracts that simultaneously achieve security-critical and gas efficiency attributes that directly impact deployment costs and vulnerability exposure in real-world blockchain environments.

This multi-dimensional preference presents a significant challenge for current LLM-based smart contract generation approaches. Security vulnerabilities in LLM-generated smart contracts pose substantial risks for blockchain deployment, as Solidity's contract-oriented nature makes it susceptible to various exploits, including reentrancy attacks [3], integer overflows [4], and access control vulnerabilities [5]. The severity of these risks was starkly demonstrated in May 2025 when Cetus Protocol, the largest decentralized exchange on the Sui blockchain, suffered a devastating exploit that drained over $260 million through smart contract vulnerabilities involving manipulated price curves and flawed reserve calculations[1]. Previous work [2] also reveals that LLMs performing well on Pass@k may produce code with elevated vulnerability rates, indicating a fundamental misalignment between model optimization objectives and human security preferences.

Equally critical is the gap in gas efficiency optimization, which represents another dimension of human preference that current LLMs largely overlook. Gas efficiency is a paramount concern for Ethereum deployment, where execution costs directly impact user adoption and economic viability [5]. While models like GPT-4 [6] and DeepSeek-R1 [7] excel at producing functionally correct contracts, they typically neglect gas consumption optimization, resulting in contracts that meet functional requirements but violate user preferences for cost-effectiveness [2], [8]. The fundamental issue lies in the misalignment between current optimization objectives and genuine human preferences for smart contracts. Optimizing for one dimension often sacrifices others [8], [9], creating a critical trade-off that is particularly problematic in decentralized finance (DeFi) applications, where high gas costs can undermine economic feasibility [10], [11]. Although recent advances leverage Supervised Fine-Tuning (SFT) and Direct Preference Optimization (DPO) for code preference alignment, existing approaches address correctness, efficiency, and security in isolation [2], [9], failing to capture the holistic preference optimization required for production-ready smart contracts that satisfy real-world user requirements [8].

To address these limitations, we introduce **PrefGen**, a novel framework that extends standard DPO beyond subjective preference pairs to incorporate quantifiable metrics that reflect genuine user priorities (i.e., gas-efficiency and security), enabling comprehensive preference-driven optimization tailored for smart contract development. Unlike existing methods that treat these objectives independently, PrefGen learns to favor solutions that minimize gas consumption while maintaining correctness and avoiding vulnerabilities through a custom

†Equal contribution.
*Corresponding authors.
Chao Ni is also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

[1]https://www.theblock.co/post/355906/cetus-poses-community-vote-to-possibly-return-100-of-funds-to-users-affected-by-223-million-exploit

DPO implementation that incorporates domain-specific reward signals. Specifically, gas efficiency rewards encourage lower consumption and security rewards penalize vulnerabilities, enabling simultaneous multi-objective optimization that aligns with real-world deployment preferences.

PrefGen employs a systematic approach to construct preference datasets by generating multiple candidate implementations for each functional requirement and evaluating them across three critical dimensions that capture real-world deployment requirements: functional correctness (i.e., Pass@k), gas efficiency (i.e., Gas@k), and security (i.e., Secure@k). This multi-dimensional evaluation framework enables the creation of preference pairs that guide the model toward generating production-ready smart contracts that simultaneously satisfy functional, gas-efficiency, and security constraints, addressing the comprehensive spectrum of developer and user preferences in blockchain environments.

Our comprehensive evaluation demonstrates that PrefGen significantly outperforms existing approaches across all three critical dimensions, bridging the gap between academic benchmarks and production deployment requirements. Compared to the pretrained Qwen-7B baseline (16.7% Pass@5, 0.0% Gas@5, 11.8% Secure@5), PrefGen achieves substantial improvements: 66.7% Pass@5, 58.9% Gas@5, and 62.5% Secure@5. Beyond these metrics, practical deployment scenarios demonstrate real-world impact: PrefGen generates ERC-20 contracts with 12% reduced gas costs compared to baseline models while completely eliminating critical security vulnerabilities such as reentrancy attacks, exemplifying the framework's ability to deliver contracts that meet the stringent requirements of production blockchain environments.

In summary, our contributions are:

- **Comprehensive Multi-Dimensional Evaluation:** We establish a systematic evaluation framework with four complementary metrics to provide a rigorous multi-dimensional assessment of generated smart contracts. Our extensive evaluation on smart contracts across 16 SOTA LLMs reveals that even top-performing models struggle with gas efficiency and security, demonstrating substantial room for improvement.
- **Novel Preference-Driven Framework:** We introduce PrefGen, a novel framework that extends DPO beyond subjective preference pairs to incorporate quantifiable blockchain-specific metrics (e.g., gas-efficiency and security).
- **Production-Ready Smart Contract Generation:** PrefGen achieves breakthrough performance improvements: 66.7% Pass@5, 58.9% Gas@5, and 62.5% Secure@5 on Qwen-7B, with practical deployment benefits including 12% reduced gas costs in ERC-20 contracts and complete elimination of critical security vulnerabilities such as reentrancy attacks. We open-source our dataset and evaluation framework at [12].

II. Background and Motivation

A. Background

1) Security Vulnerabilities in Smart Contracts: Smart contracts operate in an immutable and trustless environment where security vulnerabilities can lead to catastrophic financial losses. The critical nature of smart contract security was demonstrated in May 2025 when Cetus Protocol suffered a devastating exploit that drained over $260 million through manipulated price curves and flawed reserve calculations [13]. This incident exemplifies how implementation flaws can cascade into systemic failures, where billions of dollars in assets remain at risk. The immutable nature of blockchain deployments distinguishes smart contract security from traditional software development. Unlike conventional applications, smart contracts cannot be modified once deployed [14], [15], making prevention the only viable defense strategy. This constraint has resulted in cumulative losses exceeding billions of dollars [16]–[18], highlighting the paramount importance of generating inherently secure code.

Common vulnerability patterns include *reentrancy attacks* that exploit external calls to recursively drain funds [3], *integer overflow/underflow vulnerabilities* from unchecked arithmetic operations [4], and *access control vulnerabilities* from improper permission management [5]. Traditional static analysis tools like Slither [19] and SolTG [20] systematically study these vulnerability classes. Recent advances leverage LLMs for smart contract security analysis [21], demonstrating capabilities in identifying complex vulnerability patterns. However, these approaches focus on post-development analysis rather than prevention during generation. Critically, empirical studies reveal a fundamental misalignment: models achieving high functional correctness may simultaneously generate code with elevated vulnerability rates [2], indicating that traditional training paradigms inadequately address security considerations.

Production deployment demands simultaneous optimization across security, gas efficiency, and maintainability dimensions [22], [23]. Existing LLM-based approaches treat these objectives independently, failing to capture the holistic optimization required for production-ready contracts. This necessitates training frameworks that integrate security considerations directly into the code generation process.

2) Gas Efficiency Challenges: While LLMs have demonstrated proficiency in generating functionally correct Solidity code [1], [2], they consistently produce contracts with suboptimal gas consumption. This limitation poses barriers to practical deployment, where gas costs directly impact user adoption and economic viability.

Consider the deployment of machine learning models on Ethereum: uploading weights and test data can require 7.3×10^7 and 2.8×10^9 gas units respectively, translating to deployment costs of approximately 0.054 ETH ($13.40) and 0.2949 ETH ($737.22) [24]. These figures underscore the critical need for gas-optimized code generation in blockchain applications.

The SolEval benchmark [2] incorporates gas efficiency as an evaluation metric for repository-level Solidity generation. However, while it measures gas consumption, it lacks mechanisms for optimization. Results reveal a persistent trade-off: models achieving high functional correctness often generate contracts with prohibitive gas costs, highlighting the need for integrated optimization frameworks.

3) SFT and DPO: Supervised Fine-Tuning (SFT) and Direct Preference Optimization (DPO) represent two complementary paradigms for aligning LLMs with specific domain requirements and human preferences [25], [26]. While traditional pre-training optimizes models for next-token prediction on vast text corpora, these post-training techniques enable targeted adaptation for specialized tasks and quality criteria.

Supervised Fine-Tuning adapts pre-trained models to specific domains through supervised learning on curated instruction-response pairs [26]. In the context of code generation, SFT trains models to map natural language specifications to syntactically correct and functionally accurate implementations. This process fundamentally shifts the model's objective from general language modeling to domain-specific code synthesis, establishing a foundation for functional correctness that is essential for smart contract development.

Direct Preference Optimization addresses limitations of traditional Reinforcement Learning from Human Feedback by directly optimizing language models using preference data [25]. DPO treats the language model itself as an implicit reward model, enabling more stable and efficient preference learning. This paradigm is particularly valuable for code generation, where multiple valid implementations exist but differ significantly in quality dimensions such as efficiency, security, and maintainability [27]. For Solidity smart contract generation, these techniques address fundamental challenges that pure pre-training cannot resolve. First, smart contracts operate under unique constraints (e.g., gas costs, security vulnerabilities) and blockchain-specific semantics that are underrepresented in general programming corpora [2], [8]. SFT enables models to learn these domain-specific patterns through targeted training on Solidity implementations. Second, functional correctness alone is insufficient for production deployment [27]; contracts must simultaneously optimize for gas efficiency and security [28], [29]. DPO provides a principled framework for learning these multi-objective trade-offs through preference learning, where models learn to favor implementations that balance correctness, efficiency, and security.

Recent work has demonstrated the effectiveness of combining SFT and DPO for code generation tasks [27], [30], showing that this two-phase approach enables models to first establish functional competence through SFT, then refine quality through preference optimization. For smart contract generation, this progression is particularly critical: establishing basic Solidity syntax and semantics through SFT creates a foundation for subsequent optimization of blockchain-specific objectives through DPO.

B. Preliminary Experiment

We first conduct a preliminary experiment to evaluate the performance of LLMs on Solidity code generation. We evaluate 16 state-of-the-art LLMs on SolEval [2] with Task@k.

Results reveal significant limitations in current LLMs for smart contract generation. While the best-performing model (DeepSeek-V3) achieves only 24.99% Pass@5, indicating that even state-of-the-art LLMs struggle with functional correctness. More critically, gas efficiency remains severely compromised: the highest Gas@5 score is merely 7.13% (DeepSeek-V3), meaning over 92% of functionally correct contracts exhibit suboptimal gas consumption. Security vulnerabilities present similar challenges, with detection rates ranging from 5.28% to 19.29% across models, indicating significant room for improvement in both efficiency and security.

These findings demonstrate a fundamental trade-off: models achieving higher functional correctness (e.g., DeepSeek-V3 with 24.99% Pass@5) do not necessarily excel in gas efficiency (7.13% Gas@5) or security (19.29% Secure@5). Conversely, smaller models like GPT-4o-mini show better gas efficiency (2.42% Gas@5) but lower functional correctness (12.37% Pass@5). The results are shown in Table III.

C. Motivation

To address these fundamental limitations, we propose **PrefGen**, a framework that integrates multi-objective optimization into the training process itself. Unlike existing approaches that treat gas efficiency and security as post-hoc evaluation metrics, PrefGen incorporates these objectives directly into the learning process through extended DPO.

Our framework addresses three key gaps in current research. Rather than optimizing objectives independently, PrefGen employs preference learning to balance functional correctness, gas efficiency, and security simultaneously, representing a significant advancement in integrated optimization approaches. Furthermore, we extend standard DPO with quantifiable blockchain-specific metrics, enabling optimization tailored for smart contract deployment requirements through domain-specific rewards. Finally, PrefGen generates contracts that meet real-world deployment criteria, effectively addressing the gap between academic benchmarks and production requirements with a practical deployment focus.

III. Approach

This section presents *PrefGen*, a framework that leverages multi-objective optimization to generate functionally correct, gas-efficient, and secure Solidity smart contracts.

A. Framework Overview

As shown in Figure 1, we first construct a two-phase training pipeline that progressively refines code generation capabilities. The *SFT phase* establishes functional correctness by training on verified Solidity contracts with comprehensive test suites. The *DPO phase* then optimizes for gas efficiency and security through preference learning, where the model learns to favor solutions that minimize gas consumption while maintaining correctness and avoiding vulnerabilities.

Then, we extend standard DPO beyond human preferences to incorporate quantifiable metrics (i.e., gas consumption and security vulnerabilities), enabling multi-objective optimization tailored for smart contract development.

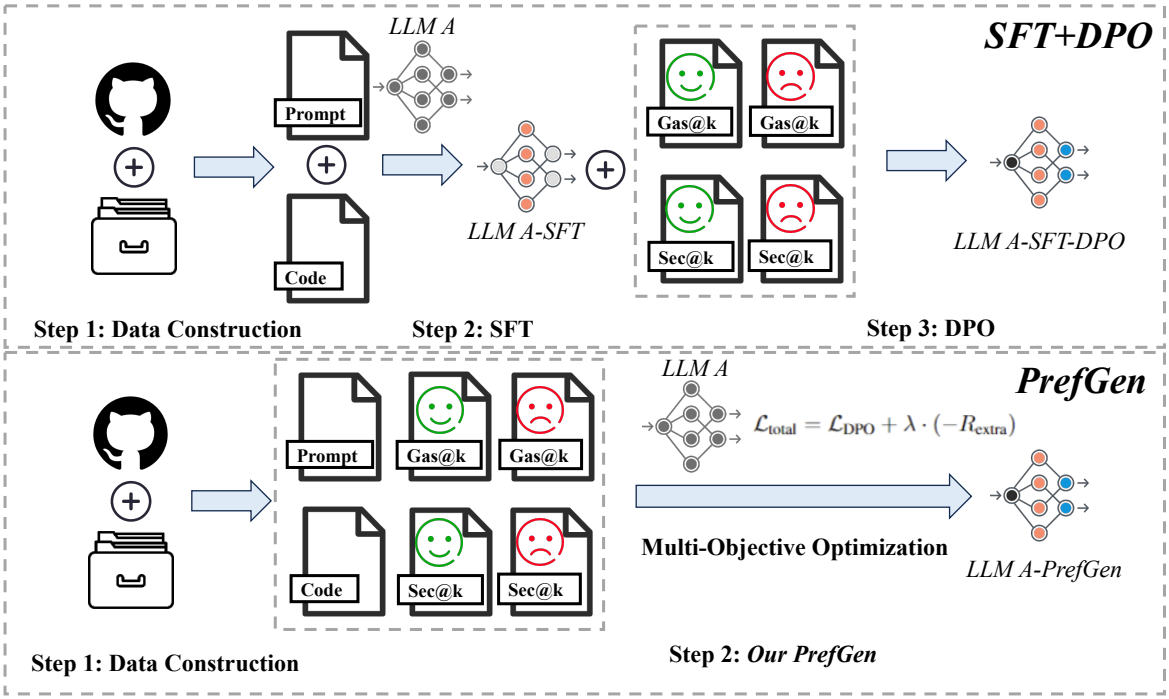

Fig. 1: Overview of the PrefGen framework v.s SFT + DPO.

B. Data Quality Ranking Algorithm

To ensure high-quality training data across both SFT and DPO phases, we employ a PageRank-inspired algorithm [31] that iteratively ranks code snippets and test cases based on mutual validation performance. This ranking mechanism is essential for identifying the most reliable training samples throughout our framework.

We initialize each code snippet and test case with a unit score. Through $M = 10$ iterations, scores are updated based on cross-validation performance:

$$\text{Score}^m(c) = (1-d) \times \text{Score}^{m-1}(c) + d \times \sum_t \text{Score}^{m-1}(t) \times \text{Link}(t,c) \quad (1)$$

$$\text{Score}^m(t) = (1-d) \times \text{Score}^{m-1}(t) + d \times \sum_c \text{Score}^{m-1}(c) \times \text{Link}(c,t) \quad (2)$$

where d is the damping factor and $\text{Link}(t,c)$ indicates whether code snippet c passes test case t. Following established PageRank practices [27], [31], we use $d = 0.85$. Our ablation analysis demonstrates this choice: at $d = 0.50$, the model achieves 63.1% Pass@5, 52.4% Gas@5, and 56.0% Secure@5; increasing to $d = 0.65$ yields 64.8%, 54.3%, 58.2%; at $d = 0.75$ produces 66.0%, 58.0%, 61.3%; the canonical $d = 0.85$ peaks at 66.7%, 58.9%, 62.5%; while $d = 0.95$ scores decline to 65.4%, 57.7%, 60.8%. This iterative refinement converges to rankings that accurately reflect code quality based on correctness validation. Higher-ranked implementations are prioritized during training, ensuring the model learns from verified, high-quality exemplars.

C. Supervised Fine-Tuning (SFT)

The SFT phase trains an LLM to generate functionally correct Solidity code from natural language specifications.

1) Dataset Construction: We carefully curate a dataset from a large-scale real-world Solidity benchmark [2], where each code sample is paired with several comprehensive test cases defining expected behavior. The data quality ranking algorithm described above is applied to select the highest-quality code-test pairs for training.

2) Training Objective: The model is fine-tuned using cross-entropy loss to minimize the divergence between predicted and ground-truth code:

$$\mathcal{L}_{\text{SFT}} = -\sum_{i=1}^{N} \log P(y_i | x_i, \theta) \quad (3)$$

where x_i represents functional specifications, y_i the corresponding Solidity implementation, and θ the model parameters. This phase optimizes the Pass@k metric, measuring the proportion of generated solutions that satisfy functional requirements.

D. Direct Preference Optimization (DPO)

The DPO phase refines the SFT model to generate gas-efficient and secure code through preference learning.

1) Preference Dataset Construction: For each functional specification, we generate multiple candidate implementations and evaluate them on Task@k metrics. Preference pairs are constructed by ranking candidates using our data quality ranking

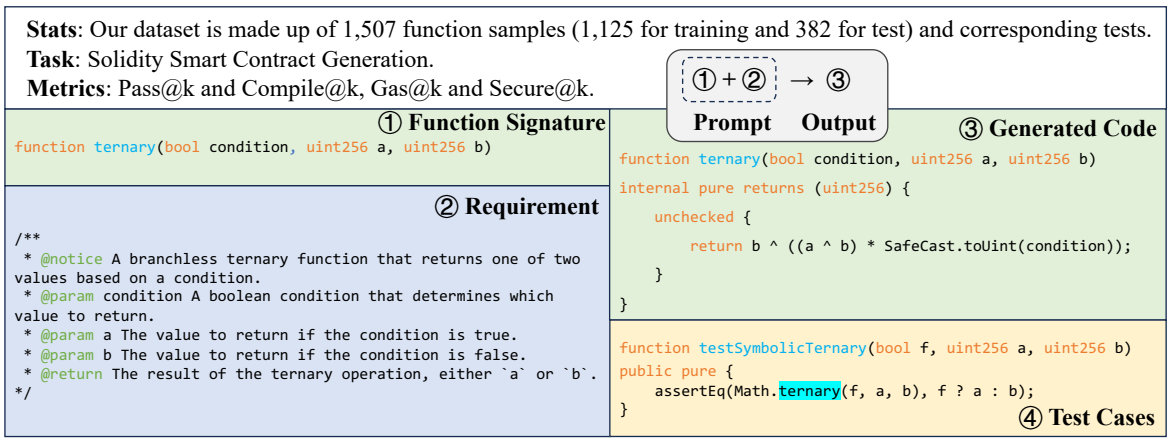

Fig. 2: Prompt construction for PrefGen.

algorithm, where solutions with higher Task@k are preferred. The details of the metrics are provided in Section IV-C.

2) Multi-Objective Optimization: We extend standard DPO to incorporate domain-specific objectives beyond human preferences. Our approach introduces additional reward terms that capture gas efficiency and security:

a) Gas Efficiency Reward::

$$R_g = -(gas_{\text{chosen}} - gas_{\text{rejected}}) \tag{4}$$

where gas_{chosen} and gas_{rejected} represent the gas consumption of the chosen and rejected solutions, respectively. This formulation encourages the model to prefer solutions with lower gas consumption by assigning positive rewards when the chosen solution is more gas-efficient.

b) Security Reward::

$$R_v = safe_{\text{chosen}} - safe_{\text{rejected}} \tag{5}$$

where $safe_{\text{chosen}}$ and $safe_{\text{rejected}}$ represent the boolean value of whether the chosen and rejected solutions are secure, respectively. This formulation encourages the model to prefer solutions that are more secure by assigning positive rewards when the chosen solution is more secure.

c) Combined Objective::

$$R_{\text{extra}} = \alpha \cdot R_g + \beta \cdot R_v \tag{6}$$

where α and β are weights of the corresponding rewards. We conduct sensitivity analysis to determine optimal values: at $\alpha = \beta = 1$, the model achieves 66.7% Pass@5, 58.9% Gas@5, and 62.5% Secure@5. For security-first settings ($\alpha = 1.2, \beta = 0.8$), performance shifts to 64.3% Pass@5, 52.1% Gas@5, and 69.8% Secure@5. Conversely, with cost-first settings ($\alpha = 0.8, \beta = 1.2$), the method yields 65.9% Pass@5, 64.2% Gas@5, and 58.7% Secure@5, demonstrating effective multi-objective trade-offs.

The final loss function integrates standard DPO with our domain-specific objectives:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{DPO}} + \lambda \cdot (-R_{\text{extra}}) \tag{7}$$

3) Implementation: We implement a custom DPO trainer extending Hugging Face's TRL library [32], overriding the loss computation to incorporate gas and security metrics. Training samples include gas consumption and vulnerability status for each preference pair, enabling simultaneous optimization across all objectives.

E. Security Integration

Security considerations are embedded throughout the framework via static analysis using Slither [19], which identifies common vulnerabilities including reentrancy attacks, integer overflows, and access control issues. During DPO training, contracts passing security checks are prioritized, ensuring the model learns to generate secure implementations.

IV. Experimental Setup

We present our constructed datasets, studied LLMs, evaluation metrics, and experimental settings.

A. Constructed Dataset

We construct our dataset following a systematic three-stage process to ensure comprehensive coverage of functional correctness, gas efficiency, and security optimization objectives. *Stage 1: Base Dataset Collection.* We utilize SolEval [2] as our foundation and expand it by systematically selecting 19 high-star GitHub projects based on specific criteria: star ratings, recent commits (within 6 months), diverse contract types (e.g., DeFi, NFT), and test case availability. We specifically filter out focal methods lacking test cases, resulting in an average of 2.4 test cases per focal function. This systematic selection yields 1,125 function samples for training and 382 samples for testing, ensuring broader coverage of real-world smart contract patterns and implementation diversity.
Stage 2: Multi-Model Code Generation. Following SolEval's best practices and the prompt construction pipeline shown in Figure 2, we perform inference across 16 mainstream large language models to generate 104,640 function samples. To handle repository-level context, we first retrieve source files and parse them with Treesitter to extract types, functions, state variables, and constants. We employ static program

analysis to identify contextual dependencies by capturing all function calls defined outside the current function scope, extracting their signatures, and building a dependency graph connecting the focal function to its required context. We store the invocation signatures along with their definitions, creating a comprehensive context database for each function sample. We then apply rigorous filtering to remove functionally incorrect implementations, retaining 12,096 valid function samples that pass all unit tests.

Stage 3: Preference-Based Sample Selection. We employ the PageRank algorithm to rank and select high-quality samples for preference optimization. This process yields our final multi-objective dataset comprising 6,520 functional-correctness optimization samples, 533 security-related samples, and 533 gas optimization samples, each containing unique problem prompts with preferred and rejected solution pairs.

TABLE I: Dataset construction statistics showing the progressive filtering and selection process.

Stage	Count
Stage 1: Base Collection	
GitHub projects crawled	19
Function samples (training)	1,125
Function samples (testing)	382
Stage 2: Code Generation	
LLMs used for inference	16
Generated function samples	104,640
Valid samples (post-filtering)	12,096
Stage 3: Final Selection	
Correctness optimization samples	6,520
Security-related samples	533
Gas optimization samples	533
Total preference pairs	**7,586**

Table I shows the dataset construction process, demonstrating the rigorous filtering and selection methodology that ensures high-quality training data across all optimization dimensions. The final dataset integrates samples from all three optimization stages, encompassing Pass@k, Gas@k, and Secure@k objectives. This multi-objective approach ensures our dataset trains models to generate contracts that are simultaneously accurate, efficient, and secure, addressing critical challenges in real-world smart contract development. We filter samples with identical or near-identical ranking scores to maintain dataset quality. We ensure no overlap between data seeds across the three optimization categories, guaranteeing diverse problem coverage and instruction variety. During training, we combine all three datasets (i.e., correctness, security, and gas) to enable simultaneous multi-objective optimization.

B. Studied Large Language Models

We select 16 state-of-the-art LLMs widely used in recent code generation studies [33]–[37]. In particular, we focus on recent models released since 2022, and we exclude the small models (with fewer than 2B parameters) due to their limited efficacy. Table II presents the state-of-the-art LLMs studied in our experiments with their sizes and types. Our experiments are based on Qwen2.5-Coder, released on November 12, 2024, ensuring evaluation with current state-of-the-art capabilities. For RQ-2 and RQ-3, we choose Qwen2.5-Coder-7B due to its strong performance in Table III, making it a promising candidate for further fine-tuning. Parameter count constraints are imposed by our available computational resources (8 × NVIDIA A800 GPUs).

TABLE II: Overview of the studied LLMs

Type	Name	Size
General LLM	DeepSeek-V3	671B (API)
	DeepSeek-R1-Distill-Qwen	7B / 32B
	DeepSeek-R1-Distill-Llama	8B
	GPT-4o	-
	GPT-4o-mini	-
	QwQ	32B
Code LLM	CodeLlama	7B / 34B
	DeepSeek-Coder	6.7B / 33B
	DeepSeek-Coder-V2-Lite	16B
	Magicoder-S-DS	6.7B
	OpenCodeInterpreter-DS	6.7B
	Qwen2.5-Coder	7B / 32B

C. Evaluation Metrics

Inspired by the famous Pass@k [38], we adopt a unified evaluation framework Task@k that measures the percentage of problems for which at least one solution among the top K samples satisfies the specific task criterion. For each problem, we generate n samples and count the number of samples c_{task} that meet the task requirement, then calculate the unbiased estimate of the probability as:

$$\text{Task@}k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c_{\text{task}}}{k}}{\binom{n}{k}} \right] \tag{8}$$

where c_{task} represents the number of solutions meeting the specific task criterion among n generated samples.

We apply this framework to evaluate four critical dimensions of smart contract generation. **Pass@k** measures functional correctness by counting solutions that pass all test cases (c_{task} = solutions passing tests). **Compile@k** evaluates syntactic correctness by counting solutions that compile successfully (c_{task} = successfully compiled solutions). **Gas@k** assesses cost optimization by counting solutions that are more gas-efficient than the reference implementation (c_{task} = gas-efficient solutions). Finally, **Secure@k** measures security robustness by counting solutions without high-risk vulnerabilities (c_{task} = secure solutions). This unified framework enables comprehensive evaluation across correctness, efficiency, and security dimensions, providing thorough assessment of smart contract generation quality while maintaining statistical rigor through the unbiased estimator from HumanEval [38].

D. Implementation

We implement our experimental pipeline using Python with PyTorch [39] and the Hugging Face library [40] for model loading and inference. Our implementation supports 16 state-of-the-art LLMs ranging from 6.7B to 671B parameters, including both general-purpose and code-specialized models.

Model Configuration. For RQ-1 baseline evaluation, we utilize models in their pretrained state, including DeepSeek-V3 (671B, accessed via API), GPT-4o and GPT-4o-mini (API access), and local models such as Qwen2.5-Coder (7B/32B), DeepSeek-Coder (6.7B/33B), CodeLlama (7B/34B), and others as detailed in Table II. For RQ-2 and RQ-3 experiments, we select Qwen-7B as our base model due to its strong performance in the baseline evaluation, making it a promising candidate for further fine-tuning under our computational constraints.

Training Hyperparameters. Our PrefGen framework employs a multi-stage training approach. For Supervised Fine-Tuning (SFT), we train for 3 epochs with a maximum input length of 2048 tokens, using a 9:1 training/validation split. For the multi-objective optimization phase, we set hyperparameters $\alpha = \beta = 1.0$ for balanced security and efficiency optimization, and $\lambda = 0.5$ for the security component based on empirical validation. All fine-tuning experiments use a batch size of 2 and a learning rate of 1e-5.

Evaluation Setup. For all experiments, we generate $n = 10$ samples per problem and evaluate using the unbiased estimator for Task@k metrics with $k \leq 10$. Temperature is set to 0.8 for diverse code generation while maintaining quality. For Pass@k evaluation, we employ Forge to conduct comprehensive fuzzing on test cases within a simulated blockchain environment, where functions failing to pass fuzzing are counted as failures in Pass@k calculation. Gas efficiency is measured using Forge execution, and security analysis employs Slither [19] for vulnerability detection.

Computational Infrastructure. Experiments are conducted on a 16-core workstation with Intel Xeon Gold 6226R CPU @ 2.90GHz, 192GB RAM, and 8×NVIDIA A800 80GB GPUs, running Ubuntu 20.04.1 LTS. RQ-1 baseline evaluation requires approximately one week of computational time, while RQ-2 framework comparison experiments take about 24 hours to reproduce. The parameter count constraints for local model evaluation ($\leq$34B) are limited by GPU memory resources.

Performance Metrics. Inference latency analysis reveals: Qwen-2.5-Coder-7B baseline requires 2.74s per sample, SFT achieves 1.46s per sample, SFT+DPO requires 1.79s per sample, while PrefGen requires 1.98s per sample. Training costs are estimated using AWS deployment: fully loaded on-demand training of Qwen-7B with PrefGen on AWS costs approximately $7.6 per epoch using p4d.24xlarge instances with 8×A100-40GB GPUs.

V. Results

To comprehensively evaluate the effectiveness of our proposed PrefGen framework, we conduct systematic experiments that address three research questions:

- **RQ-1 Empirical Evaluation.** *How do state-of-the-art LLMs perform on multi-dimensional Solidity code generation across functional correctness, gas efficiency, and security metrics?*
- **RQ-2 Framework Effectiveness Evaluation.** *How does our proposed PrefGen framework affect the performance of LLMs compared to baseline approaches including SFT, DPO, and SFT+DPO?*
- **RQ-3 Scaling Law.** *How does data scaling influence the effectiveness of LLMs?*

A. RQ-1 Empirical Evaluation

Objective. This RQ establishes a comprehensive baseline for LLM performance on Solidity code generation, evaluating state-of-the-art models across Task@k metrics. We aim to identify the current capabilities and limitations of existing LLMs in generating production-ready smart contracts.

Experimental Design. We evaluate 16 state-of-the-art LLMs on SolEval using four complementary metrics: (1) Pass@k - the percentage of solutions that pass all test cases, (2) Compile@k - the percentage of solutions that successfully compile, (3) Gas@k - the percentage of solutions that achieve optimal gas efficiency, and (4) Secure@k - the vulnerability rate in generated contracts. We group models by size categories (6.7B-16B and 32B-671B) to analyze scaling effects. All models are evaluated under identical conditions using one-shot prompting with RAG and context to ensure fair comparison.

Results. Table III presents the overall performance of state-of-the-art LLMs on SolEval. Among the 6.7B-to-16B models, DeepSeek-Coder-Lite achieves the highest Pass@1 and Compile@1, surpassing other models. Notably, DeepSeek-R1-Distill-Qwen-7B, which claims comparable performance to ChatGPT-o1-mini on benchmarks such as LiveCodeBench and CodeForces [7], underperforms compared to CodeLlama-7B. This discrepancy is likely due to DeepSeek-R1-Distill's lack of knowledge of Solidity, highlighting the importance of a specialized benchmark like PrefGen. Among the 32B-to-34B models, Qwen2.5-Coder outperforms others in both Pass@k and Compile@k. Overall, DeepSeek-V3 performs best with a 26.29% Pass@10. It is noteworthy that the distilled version of DeepSeek-R1-Qwen-32B retains significantly more of the original model's Solidity code generation capabilities during distillation compared to its 7B counterpart.

To validate PrefGen's broader applicability, we further evaluate newer models. Qwen3-8B shows limited zero-shot performance with 7.2% Pass@5, 1.7% Gas@5, and 4.3% Secure@5, which significantly improves to 28.9% Pass@5, 9.3% Gas@5, and 5.8% Secure@5 after SFT. With SFT+DPO, it achieves 26.9% Pass@5, 17.6% Gas@5, and 11.3% Secure@5, while PrefGen further enhances performance to 29.2% Pass@5, 18.9% Gas@5, and 12.1% Secure@5. Similarly, DeepSeek-R1-Distill-Qwen-7B demonstrates substantial improvements: from 4.5% Pass@5, 5.3% Gas@5, and 1.0% Secure@5 in zero-shot to 22.1% Pass@5, 7.8% Gas@5, and 3.7% Secure@5 with SFT. SFT+DPO achieves 23.8% Pass@5, 9.1% Gas@5, and 8.5% Secure@5, while PrefGen reaches 26.8% Pass@5, 14.2% Gas@5, and 12.5% Secure@5. These results consistently

TABLE III: Performance of LLMs on SolEval, evaluated using Pass@k, Compile@k, and Vulnerability Rate (Secure@k). The table presents results under the one-shot setting with RAG and Context. Bold values indicate the highest performance in each respective column. Based on the mathematical definition of Gas@k, Gas@k is always smaller than Pass@k.

LLMs	Size	Pass@1	Pass@5	Pass@10	Compile@1	Compile@5	Compile@10	Secure@5	Gas@5
6.7B to 16B (Billion) Parameters									
DeepSeek-R1-Distill-Qwen	7B	2.08%	4.50%	5.91%	6.37%	18.27%	26.29%	5.28%	0.99%
DeepSeek-R1-Distill-Llama	8B	3.67%	6.95%	8.45%	8.78%	21.68%	29.04%	6.75%	1.67%
DeepSeek-Coder-Lite	16B	**10.10%**	14.94%	16.79%	**39.44%**	**54.21%**	**57.55%**	12.27%	**4.31%**
DeepSeek-Coder	6.7B	8.39%	14.25%	16.68%	32.45%	50.74%	54.59%	**12.82%**	3.65%
CodeLlama	7B	5.15%	11.38%	14.26%	19.88%	43.05%	49.95%	10.70%	2.03%
Magicoder-S-DS	6.7B	7.26%	13.80%	16.68%	26.81%	48.77%	53.64%	12.62%	3.16%
OpenCodeInterpreter-DS	6.7B	7.05%	12.96%	15.66%	27.05%	48.71%	53.76%	11.42%	2.94%
Qwen2.5-Coder	7B	9.13%	**15.28%**	**17.44%**	33.31%	50.34%	54.44%	12.34%	4.11%
GPT-4o-mini	-	7.18%	12.37%	14.69%	38.04%	53.18%	56.66%	9.69%	2.42%
32B to 671B (Billion) Parameters									
DeepSeek-V3	671B	**21.72%**	**24.99%**	**26.29%**	**53.35%**	57.57%	58.61%	**19.29%**	**7.13%**
DeepSeek-R1-Distill-Qwen	32B	10.19%	17.06%	19.77%	31.99%	55.31%	61.31%	15.06%	3.89%
QwQ	32B	9.10%	16.74%	20.26%	48.33%	72.47%	**76.65%**	15.77%	3.68%
DeepSeek-Coder	33B	8.32%	15.57%	18.92%	29.35%	50.08%	55.39%	14.55%	3.48%
CodeLlama	34B	6.80%	13.52%	16.47%	24.59%	48.68%	54.80%	12.28%	2.75%
Qwen2.5-Coder	32B	13.46%	19.28%	21.44%	44.03%	55.53%	57.87%	16.18%	5.36%
GPT-4o	-	12.96%	20.79%	23.70%	47.04%	**58.45%**	60.74%	18.60%	4.51%

demonstrate PrefGen's effectiveness across diverse model architectures.

Regarding gas efficiency and security metrics, there is significant variation across various LLMs. DeepSeek-V3 ranks first in Pass@k but generates the most gas-inefficient contracts among the 32B-to-671B models (The higher the fee, the less efficient the codes are). Additionally, GPT-4o-mini, while being outperformed by GPT-4o in Pass@k and vulnerability rate, excels in generating contracts with lower gas fees.

Answer to RQ-1: *LLMs show varying capabilities in Solidity code generation. However, larger models do not consistently outperform smaller ones across all metrics.* ***While current LLMs can generate compilable code, they struggle with Pass@k, Gas@k, and Secure@k, indicating huge room for improvement in smart contract generation****.*

B. RQ-2 Framework Effectiveness Evaluation

Objective. This research question evaluates the effectiveness of our proposed PrefGen framework compared to traditional fine-tuning approaches. We investigate how PrefGen performs against baseline methods including pretrained models, Supervised Fine-Tuning (SFT), Direct Preference Optimization (DPO), and their combination (SFT + DPO) across functional correctness, gas efficiency, and security dimensions. Additionally, we analyze the computational efficiency trade-offs.

Experimental Design. We conduct comprehensive experiments using Qwen-7B as the base model, evaluating five configurations: (1) pretrained model without fine-tuning, (2) SFT only, (3) DPO only, (4) combined SFT + DPO, and (5) our proposed PrefGen framework. For SFT, we curate training data by filtering valid patches from 16 LLMs evaluated on SolEval, creating NL-Code pairs split 9:1 for training/validation. The model is trained for 3 epochs with 2048 token maximum input length. For DPO, we generate preference pairs based on gas and vulnerability considerations, training in two steps: first optimizing for gas, then for security. Our PrefGen incorporates a multi-objective loss function that balances correctness, gas efficiency, and security during training with hyperparameters $\alpha = \beta = 1$ for balanced optimization. All experiments use consistent evaluation metrics to ensure fair comparison.

Effectiveness Analysis. As shown in Table IV, PrefGen demonstrates superior performance across all evaluation metrics compared to traditional fine-tuning approaches. PrefGen achieves the highest Pass@5 (66.7%), significantly outperforming SFT (58.3%), DPO (25.0%), and SFT+DPO (55.7%). More notably, PrefGen excels in the challenging multi-objective optimization scenario, achieving Gas@5 of 58.9% and Secure@5 of 62.5%, substantially surpassing all baseline methods. To illustrate PrefGen's practical advantages, we conducted a targeted experiment with 100 samples comparing post-hoc filtering versus integrated training. Plain SFT produces code that passes all tests in 70 cases but satisfies both gas budget and security checks in only 8 cases. Simple post-hoc filtering (20% threshold) creates a fundamental trade-off: keeping either the most gas-efficient or the most secure outputs results in only 2 contracts remaining fully compliant across all dimensions. In contrast, PrefGen trains the model to inherently balance objectives rather than trimming outputs post-generation. After two multi-objective DPO epochs, 82 generations pass functionality tests while 65 simultaneously meet both gas and security requirements—an eight-fold improvement in deployable yield. This demonstrates that PrefGen's PageRank-guided positive/negative sampling enables models to learn multi-objective preferences simultane-

ously rather than optimizing single objectives in isolation.

TABLE IV: Performance comparison of different approaches on Qwen-7B across Pass@5, Gas@5, and Secure@5. Original refers to Pretrained (no fine-tuning).

Method	Pass@5 (%)	Gas@5 (%)	Secure@5 (%)
Original	16.7	0.0	11.8
SFT only	58.3	19.8	42.7
DPO only	25.0	6.7	17.9
SFT + DPO	55.7	37.5	48.7
PrefGen	**66.7**	**58.9**	**62.5**

Efficiency Analysis. Table V reveals that PrefGen achieves superior training efficiency despite higher memory overhead. While PrefGen requires 193.11 GB of memory compared to SFT's 76.84 GB, it completes training in only 8:53, dramatically outperforming SFT (1:22:35) and SFT+DPO (1:33:46) by 90% reduction in training time. The substantial time savings, coupled with comprehensive performance improvements across correctness, gas efficiency, and security, demonstrate PrefGen's practical viability for production smart contract development.

TABLE V: Computational efficiency comparison of different approaches on Qwen-7B. Memory usage measured in GB, training time in hours: minutes format.

Method	Memory Usage (GB)	Training Time
SFT only	76.84	1:22:35
DPO only	192.59	0:08:13
SFT + DPO	193.63	1:33:46
PrefGen	193.11	0:08:53

Real-World Application: ERC-20 and ERC-721 Contracts. Case studies with standard token contracts demonstrate PrefGen's practical effectiveness. When generating ERC-20 and ERC-721 contracts, PrefGen consistently produces functionally correct code with significant improvements in gas efficiency and security. For example, in ERC-20 token contracts, PrefGen reduces total gas costs by 12% compared to baseline approaches while eliminating vulnerabilities (i.e., reentrancy issues). Similarly, for ERC-721 marketplace contracts, PrefGen generates more gas-efficient and secure implementations than traditional fine-tuning methods.

Illustrative Code Example. To demonstrate practical improvements, consider the ERC-20 transfer function generated by different approaches as shown in Listing 1. PrefGen addresses both reentrancy vulnerabilities and gas optimization by adhering to best practices in smart contract development. By employing the "Checks-Effects-Interactions" pattern [41], PrefGen ensures that state changes are made before any external contract calls, thus preventing reentrancy attacks. Furthermore, the code reduces gas consumption by eliminating unnecessary external calls, leveraging local variable caching for balance checks, and using the 'unchecked' keyword for arithmetic operations. These optimizations enhance both security and efficiency without compromising functionality.

```
// Baseline model output (vulnerable to reentrancy)
function transfer(address to, uint256 amount) public
    returns (bool) {
    // Check sender has sufficient balance
    require(balanceOf[msg.sender] >= amount,
        "Insufficient balance");
    // Update balances before external call
    balanceOf[msg.sender] -= amount;
    balanceOf[to] += amount;
    emit Transfer(msg.sender, to, amount);
    // VULNERABIE: External call after state changes
    if (to.code.length > 0) {
        IERC20Receiver(to).onERC20Received
        (msg.sender, amount); // Reentrancy risk
    }
    return true;
}
// PrefGen  output (gas-efficient and secure)
function transfer(address to, uint256 amount) public
    returns (bool) {
    // Cache balance to save gas on repeated SLOAD
    uint256 senderBal = balanceOf[msg.sender];
    require(senderBal >= amount,
        "Insufficient balance");
    // Use unchecked block for gas optimization
    unchecked {
        // Safe subtraction (already checked above)
        balanceOf[msg.sender] = senderBal - amount;
        // Safe addition (no overflow in ERC-20)
        balanceOf[to] += amount;
    }
    emit Transfer(msg.sender, to, amount);
    // No external calls - eliminates reentrancy risk
    return true;
}
```

Listing 1: Baseline vs. PrefGen on ERC-20 functions (Yellow: vulnerable to reentrancy, Green: gas-efficient and secure)

Answer to RQ-2: *PrefGen significantly outperforms all baseline approaches (SFT, DPO, SFT+DPO) across all metrics, achieving 66.7% Pass@5, 58.9% Gas@5, and 62.5% Secure@5. While PrefGen has a moderately higher computational cost than SFT alone (approximately equivalent to DPO, lower than SFT+DPO), this efficiency trade-off is well-justified by the substantial improvements in correctness, gas efficiency, and security.* ***PrefGen effectively balances all three critical dimensions, making PrefGen highly suitable for real-world smart contract deployment.***

C. RQ-3 Scaling Law

Objective. This research question investigates the data scaling properties of PrefGen to understand how training data size affects model performance across correctness, efficiency, and security metrics. We aim to identify optimal data requirements for practical deployment and determine whether PrefGen follows typical neural scaling laws, providing guidance for resource allocation in production environments.

Experimental Design. We conduct systematic data scaling experiments using Qwen-7B, training with five different data proportions: 10%, 25%, 50%, 75%, and 100% of our curated

dataset. Each configuration maintains identical training settings (3 epochs SFT + 2 epochs DPO) with consistent hyperparameters. Data subsets are sampled uniformly to preserve the distribution of difficulty levels and contract types. We also conduct an additional experiment comparing a high-quality 25% subset (selected based on code complexity and test coverage) against a random 25% subset to assess the quality vs. quantity trade-off.

Scaling Law Analysis. As shown in Table VI, all three metrics exhibit clear scaling behaviors with increasing data size. Pass@5 improves from 21.2% with 10% data to 55.7% with the full dataset, following a logarithmic growth pattern typical of neural scaling laws. Similarly, Gas@5 shows substantial improvements from 8.7% to 37.5%, while Secure@5 increases from 9.9% to 30.5%. The rate of improvement begins to plateau beyond 75% of the data, suggesting diminishing returns as we approach the full dataset size.

TABLE VI: Performance of PrefGen with varying training data sizes on Qwen-7B across Pass@5, Gas@5, and Secure@5.

Data Size	Pass@5 (%)	Gas@5 (%)	Secure@5 (%)
10%	21.2	8.7	9.9
25% *(random)*	42.8	18.3	15.3
25% *(high-quality)*	46.1	19.8	16.7
50%	50.3	28.9	24.2
75%	53.6	34.2	29.3
100%	55.7	37.5	30.5

Efficiency vs. Data Trade-off. Interestingly, we observe that using 50% of the data achieves approximately 90% of the performance gain in Pass@5 (50.3% vs. 55.7%) while requiring only half the computational resources. This finding has practical implications for resource-constrained scenarios where training efficiency is crucial. However, for Gas@5 and Secure@5, the improvements continue to be substantial even from 75% to 100% data, indicating that gas efficiency and security optimization benefit more from larger datasets.

Data Quality vs. Quantity. To further investigate whether data quality could compensate for quantity, we conducted an additional experiment where we selected the highest-quality 25% of samples based on code complexity and test coverage. This curated subset achieved Pass@5 of 46.1%, outperforming the random 25% subset (42.8%) but still falling short of the 50% random subset. This suggests that while data quality matters, quantity remains a critical factor for achieving optimal performance in Solidity code generation.

Implications for Production Deployment. Our scaling analysis reveals several key insights for practical deployment: (1) For rapid prototyping or resource-limited scenarios, using 50% of the training data provides a good balance between performance and efficiency. (2) For production systems requiring high gas efficiency and security, the full dataset is recommended despite the additional computational cost. (3) The consistent scaling behavior across all metrics validates that PrefGen's multi-objective approach scales effectively with data, maintaining balanced improvements in correctness, efficiency, and security.

Answer to RQ-3: *PrefGen exhibits strong data scaling properties with improvements across all metrics. Notably, 50% of the data achieves approximately 90% of gain for Pass@5, offering an option for resource-constrained scenarios. However, gas efficiency and security continue to benefit substantially from larger datasets even beyond 75%.* ***While data quality can partially compensate for quantity (high-quality 25% outperforms random 25%), the results confirm that data volume remains critical for achieving optimal performance in smart contract generation.***

VI. Threats to Validity

Internal Validity. Our dataset derives from existing Solidity repositories, potentially introducing bias toward specific coding patterns. To mitigate this threat, we implement a three-stage dataset construction process (Table I) that systematically expands beyond a single source: (1) We crawl 19 diverse high-star GitHub projects spanning different domains (security, economics, games), (2) We generate 104,640 samples using 16 different LLMs to ensure implementation diversity, and (3) We employ the PageRank algorithm for preference-based sample selection, yielding 7,586 balanced preference pairs across correctness, security, and gas optimization. Gas measurements utilize Ethereum's current pricing model through Forge execution, while security analysis employs Slither [19].

External Validity. Our evaluation focuses on Ethereum-compatible Solidity contracts, potentially limiting generalizability to other blockchain platforms. To address this limitation, we design our benchmark to cover diverse contract patterns by including samples from 9 real-world repositories spanning 6 popular domains. Our RQ-1 (Table III) encompasses 16 state-of-the-art LLMs across different architectural families (general vs. code-specialized, 6.7B-671B parameters), establishing broad baseline comparisons. The repository-level evaluation approach in PrefGen simulates real-world development scenarios by providing function signatures, requirements, and repository dependencies as context.

Construct Validity. Our metrics may not capture all aspects of smart contract quality, potentially missing nuanced developer preferences. To strengthen construct validity, we implement a unified Task@k evaluation framework that extends beyond traditional Pass@k to include Compile@k, Gas@k, and Secure@k, providing a comprehensive multi-dimensional assessment. Our RQ-2 framework evaluation (Table IV) demonstrates that PrefGen achieves balanced improvements across all three dimensions (66.7% Pass@5, 58.9% Gas@5, 62.5% Secure@5), validating our multi-objective approach. We use unbiased estimators from HumanEval [38] to ensure statistical rigor ($n = 10$ samples per problem). Real-world case studies with ERC-20 and ERC-721 contracts (Listing 1) provide qualitative validation of practical improvements.

VII. Related Work

A. Code Generation Framework

The advancement of pre-training technology has significantly advanced code generation in both academia and industry [42]–

[44]. This has led to the emergence of numerous Large Language Models (LLMs) that have made substantial strides in code generation [45]–[50]. Recent advances have introduced more sophisticated models [7], [51], [52]. These models showcase the evolution from simple token prediction to understanding complex code structures and dependencies.

To optimize LLMs for various code generation scenarios, previous studies have focused on enhancing prompt engineering by introducing specific patterns, such as Structured Chain-of-Thought [53], [54], Self-planning [55], Self-debug [56], [57], and Self-collaboration [58]. However, these efforts primarily address mainstream programming languages (e.g., Java, Python, and C++) [57], [59], [60], with limited attention to domain-specific languages like Solidity. Repository-level code generation has emerged as a critical area of research. A3-CodGen [35] proposes a framework that considers local, global, and third-party library awareness for code reuse. Similarly, Shrivastava et al. [61] introduce repository-level prompt generation techniques for LLMs. These approaches underscore the critical role of contextual understanding in smart contract generation, where inter-contract interactions are prevalent. Beyond traditional benchmarks, recent evaluation frameworks have emerged. LiveCodeBench [62] addresses contamination issues in code generation evaluation by using continuously updated problems. CoderEval [36] focuses on pragmatic code generation scenarios, while Evocodebench [37] evaluates code generation in practical software projects. ClassEval [63] specifically targets class-level code generation, moving beyond simple function generation.

B. Defect Detection and Gas Fee Optimization

Smart contract security and efficiency have been extensively studied due to the immutable nature of blockchain. Chu et al. [64] provide a comprehensive survey on smart contract vulnerabilities, covering data sources, detection methods, and repair techniques. Traditional static analysis tools [19], [20], [65] have been widely adopted for identifying common vulnerability patterns in Solidity code.

Recent work has begun leveraging LLMs for smart contract analysis. Jiang et al. [21] demonstrate the potential of LLMs in identifying gas-wasting code smells in smart contracts, achieving promising results in detecting inefficient patterns that lead to unnecessary gas consumption. This approach combines the pattern recognition capabilities of LLMs with domain-specific knowledge about gas optimization.

Existing preference learning approaches for code generation, including CodeDPO [27] and SafeDPO [66], primarily focus on single objectives among functional correctness, gas efficiency, and security, lacking unified multi-objective optimization frameworks. CodeDPO and Focused-DPO improve general code correctness through preference learning but do not address the unique challenges of smart contract generation. Our work introduces a fundamental paradigm shift by incorporating quantifiable blockchain-specific rewards that enable simultaneous optimization of all three critical dimensions—functional correctness, gas efficiency, and security—within a unified preference learning framework specifically designed for smart contract generation. LLM-based approaches have shown promise in understanding complex code patterns. Studies have explored how LLMs handle ambiguous code contexts [67] and how they can be distracted by irrelevant information [68]. Jin et al. [69] propose methods to extend LLM context windows without fine-tuning, which is particularly relevant for analyzing large smart contract codebases. These insights inform the design of more effective smart contract analysis tools. Gas optimization remains critical in smart contract development, with established strategies including storage packing, loop optimization, and redundant operation elimination [28], [70], [71]. However, existing tools [70], [71] primarily focus on post-development optimization rather than generating inherently gas-efficient code. This reactive approach creates a fundamental gap that motivates the need for code generation frameworks with built-in gas efficiency considerations. Integrating defect detection and gas optimization into the code generation pipeline represents a paradigm shift from traditional sequential approaches [72], [73]. While conventional methods treat security analysis and efficiency optimization as separate post-generation phases [72], [73], incorporating these considerations directly into the generation process enables the production of inherently robust and cost-effective smart contracts. This proactive integration is crucial given the immutable nature of blockchain deployments [13]–[15], [22], [23], where vulnerabilities cannot be patched post-deployment and have led to billions of dollars in financial losses [16]–[18].

VIII. Conclusion and Future Work

This paper introduces *PrefGen*, a novel framework that simultaneously optimizes Solidity code generation for functional correctness, gas efficiency, and security through integrated Supervised Fine-Tuning and multi-objective Direct Preference Optimization. Our comprehensive evaluation demonstrates significant improvements across all metrics, achieving 66.7% Pass@5, 58.9% Gas@5, and 62.5% Secure@5, with case study results showing 12% gas cost reduction in ERC-20 contracts and elimination of reentrancy vulnerabilities. Furthermore, we extend DPO beyond human preferences to incorporate quantifiable blockchain-specific metrics, enabling production-ready smart contract generation that balances traditional trade-offs between correctness, efficiency, and security. Future work will focus on migrating our multi-objective optimization approach to other languages (e.g., Python and Java).

Acknowledgements

This work was supported in part by National Key Research and Development Program of China under Grant 2024YFB2705300, in part by the National Natural Science Foundation of China (NSFC) under Grant 62402313, in part by the Shanghai Science and Technology Innovation Action Plan under Grant 23511100400, in part by the Open Research Fund of The State Key Laboratory of Blockchain and Data Security, Zhejiang University.

REFERENCES

[1] E. Daspe, M. Durand, J. Hatin, and S. Bradai, "Benchmarking large language models for ethereum smart contract development," in *2024 6th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, 2024, pp. 1–4.

[2] Z. Peng, X. Yin, R. Qian, P. Lin, Y. Liu, C. Ying, and Y. Luo, "Soleval: Benchmarking large language models for repository-level solidity code generation," *arXiv preprint arXiv:2502.18793*, 2025.

[3] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International conference on principles of security and trust*. Springer, 2017, pp. 164–186.

[4] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[5] A. Ghaleb, J. Rubin, and K. Pattabiraman, "Achecker: Statically detecting smart contract access control vulnerabilities," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 945–956.

[6] OpenAI, "How can i access gpt-4, gpt-4 turbo, gpt-4o, and gpt-4o mini?" 2024, accessed: 2025-01-07. [Online]. Available: https://help.openai.com/en/articles/7102672-how-can-i-access-gpt-4-gpt-4-turbo-gpt-4o-and-gpt-4o-mini/

[7] DeepSeek, "Deepseek-r1," 2025, accessed: 2025-02-5. [Online]. Available: https://github.com/deepseek-ai/DeepSeek-R1

[8] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, J. Yu, Y. Wang, X. Lin, T. Chen, and Z. Zheng, "When chatgpt meets smart contract vulnerability detection: How far are we?" *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–30, 2025.

[9] P. Momeni, Y. Wang, and R. Samavi, "Machine learning-guided optimization of smart contracts," in *Proceedings of the 2019 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. IEEE, 2019, pp. 1–8.

[10] L. Gudgeon, D. Perez, D. Harz, B. Livshits, and A. Gervais, "The decentralized financial crisis," in *2020 crypto valley conference on blockchain technology (CVCBT)*. IEEE, 2020, pp. 1–15.

[11] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. Knottenbelt, "Sok: Decentralized finance (defi)," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, 2022, pp. 30–46.

[12] "Replication," 2025. [Online]. Available: https://github.com/pzy2000/PrefGen

[13] E. F. Research, "On the immutable nature of blockchain deployments and security implications," *Blockchain Security Review*, 2023, technical analysis of blockchain immutability challenges.

[14] P. Qian, R. Cao, Z. Liu, W. Li, M. Li, L. Zhang, Y. Xu, J. Chen, and Q. He, "Empirical review of smart contract and defi security: Vulnerability detection and automated repair," *arXiv preprint arXiv:2309.02391*, 2023, comprehensive survey of DeFi attacks resulting in $77.1 billion in cumulative losses.

[15] W. Li, J. Bu, X. Li, and X. Chen, "Security analysis of defi: Vulnerabilities, attacks and advances," *arXiv preprint arXiv:2205.09524*, 2022, systematic analysis of DeFi vulnerabilities and security challenges.

[16] B. S. Research, "Defi protocol financial losses analysis," 2022, documents over $3.5 billion in cumulative losses from blockchain hacks.

[17] S. Team, "Slowmist blockchain security incident database," 2024, reports cumulative losses from blockchain hacks surpassing $3.5 billion.

[18] B. R. Institute, "Analysis of smart contract security incidents and financial impact," 2023, comprehensive database of DeFi attacks and their financial consequences.

[19] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[20] K. Britikov, I. Zlatkin, G. Fedyukovich, L. Alt, and N. Sharygina, "Soltg: A chc-based solidity test case generator," in *International Conference on Computer Aided Verification*. Springer, 2024, pp. 466–479.

[21] J. Jiang, Z. Li, H. Qin, M. Jiang, X. Luo, X. Wu, H. Wang, Y. Tang, C. Qian, and T. Chen, "Unearthing gas-wasting code smells in smart contracts with large language models," *IEEE Transactions on Software Engineering*, pp. 1–26, 2024.

[22] J. J. de Leon, C. Zhang, C.-S. Koulouris, F. Medda, and Rahul, "Smart contract security in decentralized finance: Enhancing vulnerability detection with reinforcement learning," *Applied Sciences*, vol. 15, no. 11, p. 5924, 2025, addresses immutable nature of blockchain deployments and financial risks.

[23] S. Nakamoto and B. Vitalik, "Blockchain immutable nature and smart contract vulnerabilities," multiple studies confirm billions in losses due to post-deployment vulnerabilities.

[24] N. S. Sham, S. Chakraborty, and S. Sural, "Generation of optimized solidity code for machine learning models using llms," *arXiv preprint arXiv:2503.06203*, 2025.

[25] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," *Advances in Neural Information Processing Systems*, vol. 36, pp. 53 728–53 741, 2023.

[26] S. Zhang, L. Dong, X. Li, S. Zhang, X. Sun, S. Wang, J. Li, R. Hu, T. Zhang, F. Wu, and G. Wang, "Instruction tuning for large language models: A survey," *arXiv preprint arXiv:2308.10792*, 2024.

[27] K. Zhang, G. Li, Y. Dong, J. Xu, J. Zhang, J. Su, Y. Liu, and Z. Jin, "Codedpo: Aligning code models with self generated and verified source code," *arXiv preprint arXiv:2410.05605*, 2024.

[28] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 442–446.

[29] Q.-P. Kong, Z.-Y. Wang, Y. Huang, X.-P. Chen, X.-C. Zhou, Z.-B. Zheng, and G. Huang, "Characterizing and detecting gas-inefficient patterns in smart contracts," *Journal of Computer Science and Technology*, vol. 37, no. 1, pp. 67–82, 2022.

[30] X. Yin, C. Ni, L. Chen, and X. Yang, "Learning to align human code preferences," *arXiv preprint arXiv:2507.20109*, 2025.

[31] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford infolab, Tech. Rep., 1999.

[32] L. von Werra, Y. Belkada, L. Tunstall, E. Beeching, T. Thrush, N. Lambert, S. Huang, K. Rasul, and Q. Gallouédec, "Trl: Transformer reinforcement learning," https://github.com/huggingface/trl, 2020.

[33] M. A. M. Khan, M. S. Bari, X. L. Do, W. Wang, M. R. Parvez, and S. Joty, "xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval," *arXiv preprint arXiv:2303.03004*, 2023.

[34] W. Yan, H. Liu, Y. Wang, Y. Li, Q. Chen, W. Wang, T. Lin, W. Zhao, L. Zhu, S. Deng *et al.*, "Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation," *arXiv preprint arXiv:2311.08588*, 2023.

[35] D. Liao, S. Pan, X. Sun, X. Ren, Q. Huang, Z. Xing, H. Jin, and Q. Li, "A3-codgen: A repository-level code generation framework for code reuse with local-aware, global-aware, and third-party-library-aware," *IEEE Transactions on Software Engineering*, 2024.

[36] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[37] J. Li, G. Li, X. Zhang, Y. Dong, and Z. Jin, "Evocodebench: An evolving code generation benchmark aligned with real-world code repositories," *arXiv preprint arXiv:2404.00599*, 2024.

[38] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[40] S. M. Jain, "Hugging face," in *Introduction to transformers for NLP: With the hugging face library and models to solve problems*. Springer, 2022, pp. 51–67.

[41] D. Britten, V. Sjöberg, and S. Reeves, "Using coq to enforce the checks-effects-interactions pattern in deepsea smart contracts," in *FMBC 2021*, 2021.

[42] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[43] S. Shen, X. Zhu, Y. Dong, Q. Guo, Y. Zhen, and G. Li, "Incorporating domain knowledge through task augmentation for front-end javascript code generation," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1533–1543.

[44] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, S. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," in *The Eleventh International Conference on Learning Representations*, 2023.

[45] OpenAI, "Chatgpt: Optimizing language models for dialogue." 2022, accessed: 2025-01-18. [Online]. Available: https://openai.com/blog/chatgpt/

[46] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: empowering code generation with oss-instruct," in *Proceedings of the 41st International Conference on Machine Learning*, 2024, pp. 52 632–52 657.

[47] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[48] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang *et al.*, "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023.

[49] DeepSeek, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024, accessed: 2025-02-5. [Online]. Available: https://huggingface.co/papers/2401.14196

[50] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, W. Chen, and X. Yue, "Opencodeinterpreter: Integrating code generation with execution and refinement," in *Findings of the Association for Computational Linguistics ACL 2024*, 2024, pp. 12 834–12 859.

[51] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[52] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[53] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1274–1286.

[54] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–23, 2025.

[55] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao, "Self-planning code generation with large language models," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–30, 2024.

[56] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[57] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 819–831.

[58] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[59] X. Yin, C. Ni, T. N. Nguyen, S. Wang, and X. Yang, "Rectifier: Code translation with corrector via llms," *arXiv preprint arXiv:2407.07472*, 2024.

[60] X. Yin, C. Ni, X. Xu, and X. Yang, "What you see is what you get: Attention-based self-guided automatic unit test generation," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 1039–1051.

[61] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 693–31 715.

[62] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," in *The Thirteenth International Conference on Learning Representations*, 2025.

[63] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[64] H. Chu, P. Zhang, H. Dong, Y. Xiao, S. Ji, and W. Li, "A survey on smart contract vulnerabilities: Data sources, detection and repair," *Information and Software Technology*, vol. 159, p. 107221, 2023.

[65] Tree-sitter, "Tree-sitter, a parser generator tool and an incremental parsing library." 2022, accessed: 2025-01-18. [Online]. Available: https://github.com/tree-sitter/tree-sitter

[66] G.-H. Kim, Y. Jang, Y. J. Kim, B. Kim, H. Lee, K. Bae, and M. Lee, "Safedpo: A simple approach to direct preference optimization with enhanced safety," *arXiv preprint arXiv:2505.20065*, 2025.

[67] L. Kuhn, Y. Gal, and S. Farquhar, "Clam: Selective clarification for ambiguous questions with generative language models," *International Conference on Machine Learning Workshop on Deployment Challenges for Generative AI*, 2023.

[68] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. H. Chi, N. Schärli, and D. Zhou, "Large language models can be easily distracted by irrelevant context," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 210–31 227.

[69] H. Jin, X. Han, J. Yang, Z. Jiang, Z. Liu, C. Y. Chang, H. Chen, and X. Hu, "Llm maybe longlm: Selfextend llm context window without tuning," *Proceedings of Machine Learning Research*, vol. 235, pp. 22 099–22 114, 2024.

[70] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Gasol: Gas analysis and optimization for ethereum smart contracts," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018, pp. 118–125.

[71] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: analyzing the out-of-gas world of smart contracts," in *Communications of the ACM*, vol. 63, no. 10. ACM New York, NY, USA, 2020, pp. 87–95.

[72] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Computing Surveys*, vol. 53, no. 3, pp. 1–43, 2020.

[73] C. Ferreira Torres, A. K. Iannillo, A. Gervais, and R. State, "The eye of horus: Spotting and analyzing attacks on ethereum smart contracts," in *International Conference on Financial Cryptography and Data Security*. Springer, 2021, pp. 33–52.