BatCoder: Self-Supervised Bidirectional Code-Documentation Learning via Back-Translation

Anonymous ACL submission

Abstract

Training LLMs for code-related tasks typically depends on high-quality code-documentation pairs, which are costly to curate and often scarce for niche programming languages. We introduce BatCoder, a self-supervised reinforcement learning framework designed to jointly optimize code generation and documentation production. BatCoder employs a back-translation strategy: a documentation is first generated from code, and then the generated documentation is used to reconstruct the original code. The semantic similarity between the original and reconstructed code serves as an implicit reward, enabling reinforcement learning to improve the model's performance both in generating code from documentation and vice versa. This approach allows models to be trained using only code, substantially increasing the available training examples. Across Qwen2.5-Instruct models from 3B to 14B and Llama-3.1-8B, BatCoder consistently improves downstream code generation performance on HumanEval(+) and MBPP(+) over the corresponding base models. The framework is particularly effective on low-resource programming languages such as Ruby, where paired code-documentation data is scarce and the relative performance gains are substantially larger.

1 Introduction

The interaction of natural language and programming languages has been a core focus in software engineering research. Key directions include generating executable code from natural language specifications (Haiduc and Marcus, 2008; Gulwani, 2011; Chen et al., 2021) and producing natural language documentation from source code (McBurney, 2015; Jiang et al., 2017; Iyer et al., 2016). These bidirectional transformations bridge human intent and executable logic, enabling rapid prototyping and code synthesis (Austin et al., 2021; Hendrycks et al., 2021; Zhuo et al., 2024), while facilitating automated commenting and legacy code maintenance (Stolee et al., 2014; Wan et al., 2018). Overall, they enhance developer productivity and software maintainability across the development lifecycle.

Code-related tasks have been studied for over a decade. Early approaches relied on information retrieval techniques (Ye et al., 2016) as well as statistical models (Iyer et al., 2016; Wan et al., 2018). Recent advances in LLMs have substantially improved bidirectional generation between natural language and programming code. Foundation models such as GPT-5 (OpenAI, 2025), Qwen3 (Team, 2025), Gemini 2.5 (DeepMind, 2025), and Claude 3.7 (Anthropic, 2025) demonstrate strong code understanding and generation capabilities. In parallel, code-specialized models, including CodeLLaMA (Roziere et al., 2023), DeepSeek-Coder V2 (Zhu et al., 2024), and Qwen3-Coder (Team, 2025), have emerged and achieved strong performance across a wide range of code-related benchmarks (Chen et al., 2021; Austin et al., 2021; Zhuo et al., 2024).

However, training LLMs for code-documentation alignment and transformation tasks, such as code description and documentation generation, typically relies on large collections of high-quality code-documentation pairs. Despite the abundance of raw source code in public repositories such as GitHub, such paired supervision remains limited and uneven in quality, which in turn constrains the scalability and generalization of models on these tasks. To address this bottleneck, prior work has explored various data augmentation strategies. For example, WizardCoder (Luo et al., 2023) iteratively evolves instructions to construct diverse training pairs, while Magicoder (Wei et al., 2024b) synthesizes coding problems from unlabeled code snippets. Although effective, these approaches typically require stronger external models to synthesize documentation or pseudo-labels, preventing the target model from leveraging the same mechanism for self-improvement. Moreover, once such code-documentation pairs

are constructed, model optimization is usually carried out using supervised fine-tuning (SFT) or reinforcement learning (RL) paradigms, where the generated documentation is treated as fixed supervision rather than being explicitly evaluated or optimized with respect to the training objective.

To overcome these constraints, we revisit the problem from a self-supervised perspective and explore whether meaningful training signals can be derived directly from unlabeled code via model generation and feedback. A key observation is that high-quality documentation should preserve sufficient information to enable faithful reconstruction of the code, while effective code generation should adhere to the requirements described in the documentation. Consequently, the semantic similarity between the original and reconstructed code provides a natural supervisory signal for learning code-documentation transformations without explicit paired data.

Building on this insight, we propose **BatCoder**, a self-contained framework that jointly learns code description and code generation via a back-translation paradigm: given a code snippet, the model first generates documentation, which is then used to reconstruct the original code. The similarity between the original and reconstructed code serves as a unified training signal, both assessing documentation quality and guiding code generation. Leveraging recent advances in code similarity metrics such as CSSG (Xu et al., 2026), we incorporate this signal as a reward in reinforcement learning, enabling joint optimization of both stages without external documentation or teacher models.

Our contributions can be summarized as:

- We propose a self-supervised back-translation framework, named BatCoder, which enables bidirectional generation learning between code and documentation without relying on externally curated paired data, alleviating the scarcity of code-documentation supervision.
- We empirically demonstrate that BatCoder exhibits favorable scaling behavior with respect to both model capacity and training data size, showing that reconstruction-based self-supervision provides increasingly effective learning signals as scale increases.
- Through extensive experiments across multiple programming languages, we show that BatCoder consistently improves performance on both code generation and documentation production tasks, with particularly strong gains in low-resource languages.

2 Related Work

LLMs for Code. Modern LLMs, having absorbed massive collections of natural language and code during training, exhibit impressive capabilities in various coding tasks, such as code generation (Chen et al., 2021; Zhuo et al., 2024; Jain et al., 2024), code summarization (Lu et al., 2021; Sun et al., 2025; Su et al., 2025) and program repair (Xia et al., 2023; Jiang et al., 2023; Jimenez et al., 2024). Specialized models, such as CodeT5+ (Wang et al., 2023), CodeLlama (Roziere et al., 2023), DeepSeek-Coder (Guo et al., 2024), StarCoder2 (Lozhkov et al., 2024) and Qwen2.5-Coder (Hui et al., 2024), undergo dedicated pre-training or fine-tuning on code corpora to build robust code generation and understanding capabilities.

Data Augmentation for Code-Related Tasks. Despite abundant raw code data, high-quality code-text pairs for model training remain scarce, motivating data augmentation techniques to construct task-specific supervision. WizardCoder (Luo et al., 2023) iteratively evolves code-text pairs by increasing instruction complexity, incorporating constraints, and introducing adversarial elements to improve data quality. Magicoder (Wei et al., 2024b) synthesizes coding problems from unlabeled code snippets by composing code fragments and leveraging a strong LLM to ensure task coherence. Similarly exploiting unlabeled code, Gao et al. (2024) generates pseudo-labels using pre-trained models and filters low-quality instances.

Closely related to our work, UniCoder (Sun et al., 2024) also seeks to align source code with its semantic meaning by introducing structured intermediate representations, such as rule-constrained pseudo-code or documentation as a latent bridge between code understanding and generation. However, the quality of these intermediate representations is primarily ensured through hand-crafted rules or external stronger model judgments.

Another closely related work is Self-CodeAlign (Wei et al., 2024a), which also explores leveraging only the base model and unlabeled code snippets. Specifically, it extracts coding concepts from seed functions, generates instruction-response pairs via in-context learning, and validates the responses through self-generated tests and sandbox execution. However, their approach mainly serves as a data augmentation

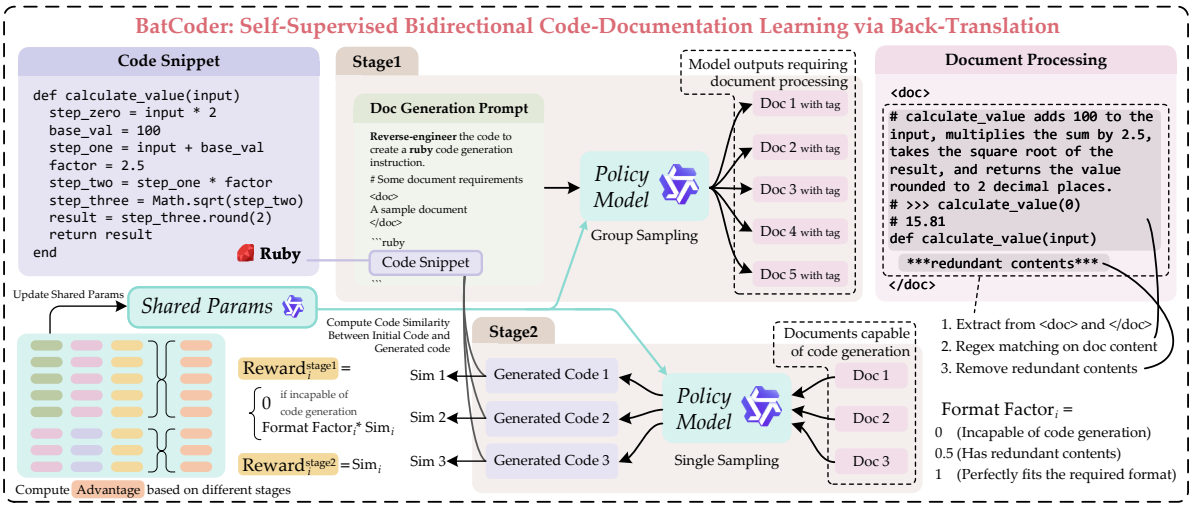


Figure 1: BatCoder training pipeline via self-supervised back-translation. Given an unlabeled code snippet c, the model first generates multiple documentation candidates in *Stage 1* (code-to-documentation). After extracting content and filtering for structural validity, each valid candidate is used to sample a single reconstructed code in *Stage 2* (documentation-to-code). Rewards consist of code similarity and document format compliance, allocated differently to the two stages. Both directions are jointly optimized via Reinforce++ algorithm.

strategy, as the model is trained using supervised fine-tuning on these execution-validated pairs.

In contrast, our method employs a self-contained bidirectional transformation between code and documentation. Rather than relying on external supervision, we evaluate the quality of generated documentation by measuring how faithfully the original code can be reconstructed through a back-translation process. This back-translation similarity then serves as the primary learning signal for both directions, enabling joint optimization without reliance on rule-based heuristics or external models.

Reinforcement Learning in Code Generation. RL enhances code generation by leveraging execution feedback to optimize model performance on various programming tasks. CodeRL (Le et al., 2022) integrates pre-trained models with actor-critic structure, employing unit test rewards to bolster functional correctness. Similarly, PPOCoder (Shojaee et al., 2023) advances this paradigm via Proximal Policy Optimization (PPO), incorporating rewards from compiler feedback, syntactic AST similarity, semantic DFG matching, and KL-divergence regularization to align generations with target codes. Ye et al. (2025) enhances code generation by applying process supervision, providing denser step-wise rewards to mitigate sparse outcome signals. Complementing these, advancements in math and code reasoning(Liu et al., 2025) synergize SFT with RL, leveraging feedback from dual domains to enhance logical capabilities.

Our framework extends these approaches by deriving rewards from code reconstruction similarity within a back-translation process, eliminating the need for target code and facilitating reward propagation to the code-to-documentation stage.

3 Methods

In this section, we present the formulation and training procedure of BatCoder. We first define the problem setting and learning objectives, followed by the sampling strategy, reconstruction-based reward design, and the reinforcement learning algorithm. BatCoder formulates code description and generation as a back-translation process, where documentation is generated from code and then used for reconstruction. Prior work shows that reconstruction similarity provides an effective signal for code-documentation alignment (Allamanis et al., 2024; Sharma, 2024). Building on this, we use the similarity between original and reconstructed code as the core signal to jointly optimize both stages. Figure 1 illustrates the overall training process.

3.1 Problem Formulation

Given a code snippet $c \in \mathcal{C}$, where $\mathcal{C}$ denotes the space of code, the BatCoder framework trains the model in two sequential stages. The transformation from code to documentation is referred to as *Stage 1*, in which the model generates a descriptive document $d = f_\theta(c) \in \mathcal{D}$, where $\mathcal{D}$ denotes the space of natural language documentation. The inverse transformation from documentation back to code is referred to as *Stage 2*, where the model

reconstructs a code snippet $c' = g_\theta(d) \in \mathcal{C}$.

The composition of *Stage 1* and *Stage 2*, formalized as $g_\theta \circ f_\theta : \mathcal{C} \to \mathcal{C}$, defines a reconstruction objective that implicitly regularizes both code generation and documentation synthesis through structural faithfulness. The model parameters θ are optimized to maximize the expected reward over the two-stage transformation:

$$J(\theta) = \mathbb{E}_{c \sim p(c)} \left[R(c, d, c') \right], \tag{1}$$

where R evaluates syntactic and semantic similarity among c, d, and c', and $p(c)$ denotes the empirical distribution of the training code corpus.

3.2 Sampling Strategy

During training, BatCoder adopts an asymmetric sampling strategy. In *Stage 1*, given an input code snippet, we sample K documentation candidates to account for the diversity of natural language descriptions. The prompts used in *Stage 1* are provided in Appendix A. Before entering *Stage 2*, the documentation generated in *Stage 1* is filtered and rewritten according to the predefined format and validity constraints. The filtering and rewriting criteria are described in Appendix B. In *Stage 2*, each selected documentation instance is used to generate only one single reconstructed code sample.

Formally, the asymmetric sampling strategy produces two sets of trajectories. In *Stage 1*, for each input code snippet c, we sample a set of documentation trajectories

$$\mathcal{T}_{\text{code2doc}}(c) = \{(c, d_k)\}_{k=1}^{K}, \tag{2}$$

Each documentation d_k is then subjected to the filtering and rewriting process. For documentation d_m that satisfy the validity constraints, a second-stage trajectory is constructed in *Stage 2* as

$$\mathcal{T}_{\text{doc2code}}(c) = \{(d_m, c'_m)\}_{m=1}^{M}, \tag{3}$$

This asymmetric design treats *Stage 1* and *Stage 2* as a single continuous rollout. For each training code snippet, we get M complete trajectories from code to documentation and back to reconstructed code. Each trajectory provides rewards for both stages, ensuring that model updates reflect the dual objectives of producing valid documentation and improving code reconstruction quality.

3.3 Reward Design

To enable self-supervised optimization in the absence of paired supervision, we design rewards that reflect both code-level similarity and documentation quality. Since code-level similarity can only be evaluated after reconstructing code from documentation, we first define the reward for *Stage 2*, and then derive the reward for *Stage 1* based on reconstruction outcomes and documentation validity.

Documentation-to-Code Reward. For *Stage 2*, the primary learning signal comes from the similarity between the original code and its reconstruction generated from the documentation. Let $\mathcal{S}(\cdot, \cdot)$ denote a code-level similarity function that measures the similarity between two code snippets. This function is abstract and can be instantiated using different structural or semantic comparison metrics. In our experiments, we instantiate $\mathcal{S}$ using CSSG (Xu et al., 2026), a code similarity metric based on an improved program dependence graph (PDG). CSSG produces scores in the range $[0, 1]$, where higher values indicate stronger semantic and structural similarity between code snippets. We further compare the impact of different similarity metrics in the ablation study.

For each trajectory $(d_m, c'_m) \in \mathcal{T}_{\text{doc2code}}(c)$ generated in *Stage 2*, we define a documentation-to-code similarity reward by comparing the reconstructed code with the original input code c:

$$R^{(m)}_{\text{doc2code}} = R^{(m)}_{\text{sim,doc2code}} = \mathcal{S}(c, c'_m). \tag{4}$$

Code-to-Documentation Reward. For each trajectory $(c, d_k) \in \mathcal{T}_{\text{code2doc}}(c)$ generated in *Stage 1*, we compute a similarity-based reward by comparing the original code with its reconstruction:

$$R^{(k)}_{\text{sim,code2doc}} = R^{(k)}_{\text{sim,doc2code}} = \mathcal{S}(c, c'_k). \tag{5}$$

In addition to reconstruction similarity, we incorporate a documentation validity reward $R^{(k)}_{\text{doc}}$. Beyond reflecting the structural correctness and well-formedness of the generated documentation, this reward is designed to stabilize the documentation generation during training by encouraging consistent adherence to the required format and reducing variance in early-stage optimization. The auxiliary validity reward is defined as:

$$R^{(k)}_{\text{doc}} = \begin{cases} 0, & \text{incapable of code generation,} \\ 0.5, & \text{contains redundant content,} \\ 1, & \text{perfectly fits the required format.} \end{cases} \tag{6}$$

The final reward for *Stage 1* is defined at the trajectory level:

$$R^{(k)}_{\text{code2doc}} = R^{(k)}_{\text{sim,code2doc}} \cdot R^{(k)}_{\text{doc}}. \tag{7}$$

where $R^{(k)}_{\text{sim,code2doc}}$ is only computed when a valid reconstruction c'_k is available.

In practice, the documentation generated in *Stage 1* is first subjected to the filtering and rewriting process described in Section 3.2. If a documentation instance d_k fails to satisfy the predefined validity constraints, it is excluded from the subsequent documentation-to-code stage and no reconstruction is performed. Such trajectories terminate at (c, d_k) and do not contribute to the *Stage 2* optimization, and the code-to-documentation reward is explicitly defined as $R^{(k)}_{\text{code2doc}} = 0$.

3.4 Reinforcement Learning Algorithm

To optimize BatCoder, we adopt Reinforce++ (Hu, 2025), a policy gradient algorithm that supports on-policy updates for improved sample efficiency. Training proceeds over trajectories generated by the code-documentation back-translation process described in Section 3.2. The reward for the *Stage 2*, denoted as R_{doc2code}, and the reward for the *Stage 1*, denoted as R_{code2doc}, are computed as defined in Section 3.3. Samples generated during training are stored in a fixed-size replay buffer $\mathcal{B}$, including code-to-documentation pairs (c, d_k) with their corresponding rewards $R^{(k)}_{\text{code2doc}}$, and documentation-to-code pairs (d_m, c'_m) with rewards $R^{(m)}_{\text{doc2code}}$.

Model parameters θ are updated by sampling minibatches from $\mathcal{B}$. To stabilize training, rewards are normalized using statistics computed over the buffer, yielding the following token-level advantage estimates:

$$A^{(k)}_{\text{code2doc}} = \frac{R^{(k)}_{\text{code2doc}} - \mu_{\text{code2doc}}}{\sigma_{\text{code2doc}}}, \tag{8}$$

$$A^{(m)}_{\text{doc2code}} = \frac{R^{(m)}_{\text{doc2code}} - \mu_{\text{doc2code}}}{\sigma_{\text{doc2code}}}, \tag{9}$$

where $\mu_{\text{code2doc}}, \mu_{\text{doc2code}}$ and $\sigma_{\text{code2doc}}, \sigma_{\text{doc2code}}$ denote the mean and standard deviation of the corresponding rewards maintained in $\mathcal{B}$.

The training objective jointly optimizes both generation directions. Specifically, the loss function is defined as follows:

$$\begin{aligned} L(\theta) = &-\mathbb{E}_{\mathcal{T}\sim\mathcal{B}}\big[A^{(k)}_{\text{code2doc}} \cdot \log \pi_\theta(d_k \mid c)\big] \\ &- \mathbb{E}_{\mathcal{T}\sim\mathcal{B}}\big[A^{(m)}_{\text{doc2code}} \cdot \log \pi_\theta(c'_m \mid d_m)\big] \\ &+ \beta \cdot \mathrm{KL}(\pi_\theta \,\|\, \pi_{\text{ref}}), \end{aligned} \tag{10}$$

where the KL regularization term constrains updates with respect to a reference policy π_{ref}. The overall procedure is summarized in Appendix C.

4 Experimental Setup

4.1 Base Models and Training Data

Qwen2.5-Instruct models at 3B, 7B, and 14B scales (Team, 2024), together with Llama-3.1-8B (Grattafiori et al., 2024), are used as base models. These models are widely adopted in recent code-related studies and allow us to evaluate the robustness of our approach across different model families and scales. All experiments are initialized from the corresponding pretrained checkpoints released on the Hugging Face platform.

For training, we used the code data from the Code-Text task in the CodeXGLUE benchmark suite[1] (Husain et al., 2019; Lu et al., 2021). This dataset contains code samples from multiple programming languages, including Python, Ruby, and Go, making it well suited for training on a unified corpus and evaluating generalization across languages. Although the original dataset includes accompanying text fields, these texts primarily consist of function-level docstrings rather than the structured documentation targeted in our setting. Consequently, we discarded the text component and used only the code snippets during training. Further implementation details, including full training configurations, are deferred to Appendix D.

4.2 Baselines

We compare BatCoder against a diverse set of representative baselines, including widely adopted code-oriented LLMs that are pre-trained or fine-tuned on large-scale code corpora. The open-source baselines include CodeT5+ (Wang et al., 2023), CodeLlama (Roziere et al., 2023), WizardCoder (Luo et al., 2023), Magicoder (Wei et al., 2024b), StarCoder2 (Lozhkov et al., 2024), DeepSeek-Coder-Instruct (Guo et al., 2024), Qwen2.5-Instruct (Hui et al., 2024) and Llama-3.1 (Grattafiori et al., 2024). In addition, we report results from two closed-source models, GPT-4o and O1 Mini, which represent the state-of-the-art among contemporaneous models on the EvalPlus leaderboard[2] (Liu et al., 2023) and serve as upper-bound references. All results are consistently drawn from the leaderboard.

4.3 Evaluation Setup

We evaluate model performance mainly on HumanEval (Chen et al., 2021) and MBPP (Austin

[1] https://huggingface.co/datasets/google/code_x_glue_ct_code_to_text

[2] https://evalplus.github.io/leaderboard.html

Table 1: Pass@1 (%) results on HumanEval(+) and MBPP(+). Results for Qwen2.5-Instruct, Llama-3.1 and BatCoder are obtained using the `evalplus` framework, while all other baseline results are retrieved from the EvalPlus Leaderboard. BatCoder consistently improves upon its base model and surpasses open-source baselines with comparable or even larger parameter scales.

Scale	Model	Size	Benchmark		Open-Source
			HumanEval (+)	MBPP (+)	
Unknown	GPT-4o	Unknown	92.7 (87.2)	87.6 (72.2)	×
	O1 Mini	Unknown	**96.3** (**89.0**)	**93.1** (**78.8**)	×
≥14B	CodeT5+	16B	31.7 (26.8)	56.6 (47.1)	✓
	StarCoder2	15B	46.3 (37.8)	78.0 (65.1)	✓
	CodeLlama-Instruct	34B	51.8 (43.9)	69.3 (56.3)	✓
	WizardCoder-Python	34B	73.2 (64.6)	75.1 (63.2)	✓
	DeepSeek-Coder-Instruct	33B	81.1 (75.0)	80.4 (70.1)	✓
	Qwen2.5-Instruct	14B	82.3 (73.8)	81.7 (69.3)	✓
	BatCoder-Qwen2.5-Instruct	14B	**85.4** (**78.0**)	**84.9** (**71.7**)	✓
6∼14B	Llama-3.1	8B	68.3 (61.6)	66.9 (54.5)	✓
	BatCoder-Llama-3.1	8B	70.1 (62.2)	69.6 (57.7)	✓
	Magicoder-S-DS	6.7B	76.8 (71.3)	79.4 (69.0)	✓
	Qwen2.5-Instruct	7B	81.7 (73.2)	78.6 (68.0)	✓
	BatCoder-Qwen2.5-Instruct	7B	**84.1**(**77.5**)	**81.5** (**69.6**)	✓
≤3B	StarCoder2	3B	31.7 (27.4)	57.4 (47.4)	✓
	DeepSeek-Coder-Instruct	1.3B	65.9 (60.4)	65.3 (54.8)	✓
	Qwen2.5-Instruct	3B	73.8 (68.3)	73.0 (62.2)	✓
	BatCoder-Qwen2.5-Instruct	3B	**76.2** (**71.3**)	**75.9** (**66.4**)	✓

et al., 2021), two widely adopted benchmarks for assessing code generation quality. HumanEval and MBPP consists of Python programming problems with unit tests for execution-based evaluation. To more thoroughly assess model performance, we further report results on HumanEval+ and MBPP+, the rigorized versions from EvalPlus (Liu et al., 2023) that augment the original benchmarks with 80× and 35× more test cases.

In addition to Python-centric evaluation, we further examine model performance on programming languages with comparatively smaller training corpora, where the scarcity of paired code-documentation data is more pronounced. To this end, we include MultiPL-E (Cassano et al., 2022) for evaluation and focus on Ruby and Go, which are covered in the CodeXGLUE training data. MultiPL-E extends HumanEval and MBPP by providing translated versions of them in other programming languages. Evaluation details and additional benchmark results are provided in Appendix E.

5 Results and Discussion

5.1 Python Code Generation

Table 1 summarizes the main results on HumanEval, HumanEval+, MBPP, and MBPP+. The consistent improvements across the Qwen2.5-Instruct 3B, 7B, and 14B scales indicate that BatCoder generalizes well across different model capacities, rather than being tailored to a specific parameter regime. Notably, the 7B BatCoder model outperforms both the larger Qwen2.5-14B-Instruct base model and the substantially larger DeepSeek-Coder-33B-Instruct on HumanEval and HumanEval+, while also achieving comparable or better performance on MBPP and MBPP+, demonstrating the effectiveness of the proposed self-supervised training signal beyond merely increasing model size. Similar improvements are also observed on Llama-3.1-8B, where BatCoder improves MBPP pass@1 from 66.9 to 69.6 and MBPP+ from 54.5 to 57.7, suggesting that the proposed framework is applicable across different model families and architectures.

5.2 Multilingual Code Generation

We further conducted experiments on MultiPL-E for Ruby and Go, whose high-quality code-documentation resources are comparatively scarce. Obtaining paired data for these languages is more challenging than Python under conventional training paradigms, and the low baseline scores suggest that the base models may have limited exposure to these languages during training. This setting provides a meaningful testbed for examining whether BatCoder can not only strengthen existing code capabilities but also elicit non-trivial performance in

Table 2: Pass@1 results on MultiPL-E for Ruby and Go. We compare the base Qwen2.5-Instruct models with BatCoder at different model scales, observing consistent and substantial improvements across both programming languages and model sizes.

Model	Size	Ruby	Go
Qwen2.5-Instruct	7B	3.1	34.4
BatCoder (Ours)	7B	**13.0**	**39.0**
Qwen2.5-Instruct	3B	0.0	33.8
BatCoder (Ours)	3B	**10.6**	**37.7**

Table 3: Pass@1 ablation results on Ruby from MultiPL-E. We compare SFT, partial variants of BatCoder, and the full framework, with the full model achieving the best performance.

Training Setting	Pass@1
Base Model	0.0
SFT	6.2
BatCoder (w/o *Stage 1*)	1.9
BatCoder	**10.6**

previously underrepresented languages.

Table 2 reports the evaluation results. BatCoder consistently outperforms the corresponding Qwen2.5-Instruct baselines across both Ruby and Go, with markedly larger gains on Ruby. In particular, at the 3B scale, the base model attains a pass@1 of 0.0 on Ruby, indicating a complete failure to solve any test instances, whereas BatCoder raises performance to 10.6. This substantial jump from zero to a non-trivial accuracy highlights the effectiveness of the proposed framework in extremely low-resource regimes. A similar but less extreme pattern is observed at the 7B scale on Ruby, where BatCoder improves pass@1 from 3.1 to 13.0, yielding an absolute gain of nearly 10 points. On Go, where the base models already exhibit moderate performance, BatCoder still delivers consistent improvements at both model scales, increasing pass@1 from 33.8 to 37.7 for the 3B model and from 34.4 to 39.0 for the 7B model.

Taken together, these results demonstrate that BatCoder generalizes effectively across model scales and is particularly well suited to low-resource programming languages, suggesting promising applicability to a broader range of low-resource code domains.

5.3 Ablation Study

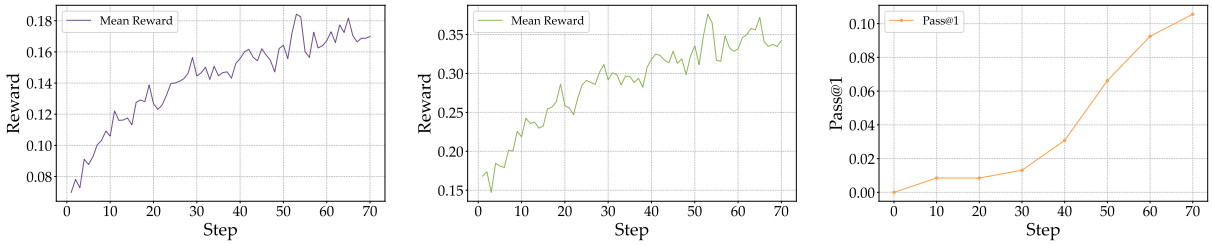
We conduct ablation experiments on Ruby from MultiPL-E to better understand the contribution of different components in BatCoder, focusing on the training-stage design, particularly the role of *Stage 1* and its associated reward. The base model used in these experiments is Qwen2.5-3B-Instruct. Additional ablations on the choice of similarity functions are provided in Appendix F.

Effect of the Code-to-Documentation Stage We examine the impact of optimizing documentation generation by disabling parameter updates associated with *Stage 1*, while preserving *Stage 2* optimization based on code-level similarity. In this setting, the model still produces documentation, but it is no longer guided by an explicit reinforcement signal and does not contribute to parameter updates. As reported in Table 3, this variant leads to only a marginal improvement over the base model, with pass@1 increasing from 0.0 to 1.9, and remains far below the performance achieved by the complete BatCoder. This observation highlights the importance of directly optimizing documentation generation. When documentation is not encouraged to preserve sufficient information for faithful code reconstruction, the resulting self-supervised signal becomes substantially weaker, limiting the effectiveness of subsequent code generation optimization. To illustrate the differences between documentation generated before and after BatCoder training, we provide a case study in Appendix G.

Comparison with Supervised Fine-Tuning. We further compare BatCoder against a SFT baseline to assess whether the observed gains can be attributed solely to exposing the model to additional synthetic data. In this setting, documentation is generated by the base model, with the same filtering and rewriting procedures applied as in BatCoder, and resampling performed until valid documents are obtained. The model is then fine-tuned using these documents as inputs and the original code as targets, following a conventional SFT paradigm. While SFT improves performance over the base model, it remains notably inferior to the complete BatCoder. This suggests that training on model-generated pseudo pairs is insufficient. Instead, explicitly evaluating and reinforcing documentation quality through code-level similarity provides a more effective learning signal.

5.4 Documentation Quality During Training

To evaluate how documentation quality evolves during training, we conduct a longitudinal analysis on a held-out Python subset of CodeXGLUE, using



(a) *Stage 1* mean reward curve (b) *Stage 2* mean reward curve (c) Pass@1 of MultiPL-Ruby

Figure 2: Training dynamics of BatCoder. All curves show a consistent upward trend, indicating that the reward signals are aligned with the training objectives and correlate with improved model performance.

Qwen2.5-3B-Instruct as the base model. At different training stages, each model generates 500 documentation samples, which are evaluated by GPT-5.4 along three dimensions: Function Description Clarity, Example Code Correctness, and Readability (0–5 scale). The evaluation prompt and scoring criteria are provided in Appendix H.

We report results at every 10 training steps until convergence (approximately 34 steps), as summarized in Table 4.

Steps	Clarity	Correctness	Readability	Total
0	1.56	0.94	2.27	4.77
10	1.55	0.94	2.35	4.84
20	1.58	0.97	2.38	4.93
30	1.60	1.00	2.39	4.99
34	**1.60**	**1.01**	**2.41**	**5.02**

Table 4: Documentation quality during training. Step 0 corresponds to the base model. Results show consistent improvements in overall quality over time.

We observe a consistent upward trend in readability and overall quality, with only minor fluctuations typical of RL. These results indicate that optimizing for reconstruction-based rewards leads to steady improvements over training.

5.5 Training Dynamics and Reward Analysis

We further analyze the training dynamics focusing on the evolution of reward signals and downstream performance. We track the mean rewards for *Stage 1* and *Stage 2*, together with Pass@1 performance on MultiPL-Ruby. All curves are obtained from BatCoder-3B trained on the Ruby subset, and are summarized in Figure 2.

As shown in Figures 2a and 2b, the mean rewards for both stages increase steadily as training progresses, exhibiting smooth upward trends. Figure 2c reports the corresponding Pass@1 scores on MultiPL-Ruby, which also improve consistently over training steps and follow a similar progression pattern to the reward curves.

Overall, the consistent alignment between reward trajectories and downstream evaluation performance suggests that the proposed reward design provides a stable and meaningful optimization signal. The bidirectional RL objective effectively guides both documentation generation and code reconstruction, resulting in tangible improvements in code generation quality. Although evaluated at a fixed training scale, the observed training dynamics indicate that the framework has the potential to further benefit from larger training data.

6 Conclusion

We present **BatCoder**, a self-supervised framework that jointly learns code generation and documentation synthesis through a back-translation paradigm. By enforcing consistency between code and documentation under bidirectional transformations, BatCoder enables effective optimization directly from unlabeled code, without requiring paired supervision or external teacher models.

Experimental results show that BatCoder consistently improves both code generation and documentation quality across multiple benchmarks, outperforming supervised and synthetic-data-based baselines at comparable model scales. The observed gains generalize across different model capacities and architectures, including both Qwen and Llama families. Notably, the proposed framework is particularly effective in low-resource programming languages, such as Ruby and Go, where curated code-documentation pairs are scarce. These findings indicate that back-translation similarity provides a robust learning signal in data-constrained settings, facilitating meaningful improvements from unlabeled code alone. The future directions are discussed in Appendix I.

Limitations

While BatCoder demonstrates strong empirical performance across multiple benchmarks and programming languages, we acknowledge several limitations that point toward promising avenues for future work.

First, the current framework relies on reconstruction-based similarity as the primary optimization signal. Although this objective provides an effective form of self-supervision, it mainly captures consistency between generated documentation and reconstructed code, rather than directly modeling broader software engineering properties such as long-range program behavior, execution efficiency, or maintainability. Incorporating richer forms of feedback may further improve the quality of both code generation and documentation synthesis.

Second, BatCoder is evaluated primarily on function-level generation tasks. While this setting provides a controlled environment for studying bidirectional code-documentation learning, real-world software development often involves repository-level context, cross-file dependencies, and interaction with external libraries or APIs. Extending the framework toward larger-scale software engineering scenarios remains an important direction for future work.

Finally, although the proposed framework exhibits favorable scaling behavior across model sizes and architectures, reinforcement learning over long generation trajectories still introduces non-trivial computational overhead compared with conventional supervised fine-tuning. Improving training efficiency and stability for larger-scale settings remains an important practical consideration.

Acknowledgments

This document has been adapted by Steven Bethard, Ryan Cotterell and Rui Yan from the instructions for earlier ACL and NAACL proceedings, including those for ACL 2019 by Douwe Kiela and Ivan Vulić, NAACL 2019 by Stephanie Lukin and Alla Roskovskaya, ACL 2018 by Shay Cohen, Kevin Gimpel, and Wei Lu, NAACL 2018 by Margaret Mitchell and Stephanie Lukin, BibTEX suggestions for (NA)ACL 2017/2018 from Jason Eisner, ACL 2017 by Dan Gildea and Min-Yen Kan, NAACL 2017 by Margaret Mitchell, ACL 2012 by Maggie Li and Michael White, ACL 2010 by Jing-Shin Chang and Philipp Koehn, ACL 2008 by Johanna D. Moore, Simone Teufel, James Allan, and Sadaoki Furui, ACL 2005 by Hwee Tou Ng and Kemal Oflazer, ACL 2002 by Eugene Charniak and Dekang Lin, and earlier ACL and EACL formats written by several people, including John Chen, Henry S. Thompson and Donald Walker. Additional elements were taken from the formatting instructions of the *International Joint Conference on Artificial Intelligence* and the *Conference on Computer Vision and Pattern Recognition*.

References

Miltiadis Allamanis, Sheena Panthaplackel, and Pengcheng Yin. 2024. Unsupervised evaluation of code llms with round-trip correctness. In *Proceedings of the 41st International Conference on Machine Learning*, pages 1050–1066.

Anthropic. 2025. Claude 3.7 Sonnet.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. 2022. A framework for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, and 1 others. 2022. Multiple: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Google DeepMind. 2025. Gemini 2.5.

Shuzheng Gao, Wenxin Mao, Cuiyun Gao, Li Li, Xing Hu, Xin Xia, and Michael R Lyu. 2024. Learning in the wild: Towards leveraging unlabeled data for effectively tuning pre-trained code models. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 969–981. IEEE.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Sonia Haiduc and Andrian Marcus. 2008. On the use of domain terms in source code. In *2008 16th IEEE International Conference on Program Comprehension*, pages 113–122. IEEE.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Jian Hu. 2025. Reinforce++: A simple and efficient approach for aligning large language models. *arXiv preprint arXiv:2501.03262*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *54th Annual Meeting of the Association for Computational Linguistics 2016*, pages 2073–2083. Association for Computational Linguistics.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442. IEEE.

Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–146. IEEE.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *12th International Conference on Learning Representations, ICLR 2024*.

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

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Zihan Liu, Zhuolin Yang, Yang Chen, Chankyu Lee, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. 2025. Acereason-nemotron 1.1: Advancing math and code reasoning through sft and rl synergy. *arXiv preprint arXiv:2506.13284*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. Starcoder 2 and the stack v2: The next generation. *ArXiv*, abs/2402.19173.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, and 1 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Paul W McBurney. 2015. Automatic documentation generation via source code summarization. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 903–906. IEEE.

OpenAI. 2025. Gpt-5 system card.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 311–318.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *Preprint*, arXiv:2009.10297.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Asankhaya Sharma. 2024. Patched rtc: evaluating llms for diverse software development tasks. *arXiv preprint arXiv:2407.16557*.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*.

Kathryn T Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):1–45.

Chia-Yi Su, Aakash Bansal, Yu Huang, Toby Jia-Jun Li, and Collin McMillan. 2025. Context-aware code summary generation. *Journal of Systems and Software*, page 112580.

Tao Sun, Linzheng Chai, Jian Yang, Yuwei Yin, Hongcheng Guo, Jiaheng Liu, Bing Wang, Liqun Yang, and Zhoujun Li. 2024. Unicoder: Scaling code large language model via universal code. *arXiv preprint arXiv:2406.16441*.

Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2025. Source code summarization in the era of large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1882–1894. IEEE.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 397–407.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. 2024a. Selfcodealign: Self-alignment for code generation. *Advances in Neural Information Processing Systems*, 37:62787–62874.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024b. Magicoder: empowering code generation with oss-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, pages 52632–52657.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE Computer Society.

Jingwen Xu, Yiyang Lu, Changze Lv, Zisu Huang, Zhengkang Guo, Zhengyuan Wang, Muzhao Tian, Xuanjing Huang, and Xiaoqing Zheng. 2026. Cssg: Measuring code similarity with semantic graphs. *arXiv preprint arXiv:2601.04085*.

Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th international conference on software engineering*, pages 404–415.

Yufan Ye, Ting Zhang, Wenbin Jiang, and Hua Huang. 2025. Process-supervised reinforcement learning for code generation. *arXiv preprint arXiv:2502.01715*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Terry Yue Zhuo, Armel Zebaze, Nitchakarn Suppattarachai, Leandro von Werra, Harm de Vries, Qian Liu, and Niklas Muennighoff. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Prompt Templates

This appendix presents the prompt templates used for documentation sampling in *Stage 1*. The prompts are designed to elicit structured and well-formed documentation that can be reliably used for subsequent reconstruction in *Stage 2*. We adopt

a one-shot prompting strategy, providing a single format-aligned example to encourage the model to consistently follow the desired documentation structure.

A.1 Python Documentation Generation Prompt

```
Please analyze the code provided at the
end and reverse-engineer it to create a
python code generation instruction.
The output must be wrapped in `<doc>`
and `</doc>` tags and include:

1. Any necessary library imports.
2. The Python function definition line
with type annotations.
3. Indented by 4 spaces, a docstring
starting with `"""` that includes a
description of what the code does.
4. Indented by 4 spaces, one or several
illustrative input/output examples (
using `>>>` syntax) inside the
docstring.
5. Indented by 4 spaces, the closing
`"""` of the docstring.

Here is an example of the expected
structure (pay attention to the format,
not the content):

<doc>
import json
def test_func(arg1: str, arg2: int) ->
bool:
    """ test_func implements the
functionality of ...
    >>> test_func('example', 1)
    True
    """
</doc>

Now, please generate the python code
problem statement within `<doc>` and `</
doc>`:
```{origin_lan}
{code}
```
```

A.2 Ruby Documentation Generation Prompt

```
Please analyze the code provided at the
end and reverse-engineer it to create a
ruby code generation instruction.
The output must be wrapped in `<doc>`
and `</doc>` tags and include:

1. Any necessary `require` statements.
2. A descriptive problem statement
comment explaining what the code does.
3. One or several illustrative input/
output comment line (using `>>>` syntax)
.
4. The Ruby function definition line
matching the logic (needs to complete
generating the leading spaces of the
next line).

Here is an example of the expected
structure (pay attention to the format,
not the content):

<doc>
require 'json'
# test_func implements the functionality
 of ...
# >>> test_func(arg1, arg2)
# expected_result
def test_func(arg1, arg2)
  </doc>

Now, please generate the ruby code
problem statement within `<doc>` and `</
doc>`:
```{origin_lan}
{code}
```
```

A.3 Go Documentation Generation Prompt

```
Please analyze the code provided at the
end and reverse-engineer it to create a
go code generation instruction.
The output must be wrapped in `<doc>`
and `</doc>` tags and include:

1. The package declaration and relevant
imports.
2. A descriptive problem statement
comment explaining what the code does.
3. One or several illustrative input/
output comment line (using `>>>` syntax)
.
```

```
4. The Go function definition line
matching the logic (needs to complete
generating the leading tab of the next
line).

Here is an example of the expected
structure (pay attention to the format,
not the content):

<doc>
package main
import "fmt"
// test_func implements the
functionality of ...
// >>> test_func(arg1, arg2)
// expected_result
func test_func(arg1 Type, arg2 Type)
Type {
        </doc>

Now, please generate the go code problem
 statement within `<doc>` and `</doc>`:
```{origin_lan}
{code}
```
```

B Filtering and Rewriting

This appendix describes the filtering and rewriting procedure applied to the documentation generated in *Stage 1* before it is passed to *Stage 2*. The purpose of this process is to retain only structurally valid and semantically informative documentation, ensuring that reconstruction-based rewards are computed on reliable inputs under the back-translation setting.

B.1 Documentation Structure Constraints

To enforce a consistent documentation format, we apply a set of structural constraints using regular-expression-based matching. These constraints are aligned with the prompt specification and the one-shot example provided in Appendix A, ensuring consistency between the expected documentation format and the filtering criteria.

A documentation instance is considered valid if it contains: (i) one or more natural language description lines summarizing the functionality, (ii) at least one illustrative input-output example, and (iii) a function or procedure definition corresponding to the described behavior. These constraints are designed to be language-agnostic and capture common documentation patterns across programming languages.

B.2 Filtering and Rewriting Procedure

The filtering procedure consists of three steps. First, only the content enclosed within designated documentation tags (e.g., <doc> and </doc>) is extracted from the raw model output. Generations without valid tags are discarded. Second, the extracted content is matched against the structural constraints described above. If a valid match is found, the documentation is truncated at the end of the matched span, removing any trailing or extraneous content beyond the required structure. This rewriting step is applied only when additional text appears after the end of the structurally valid documentation. In such cases, the instance is marked as containing redundant content, and only the prefix that satisfies the structural constraints is retained.

In addition to producing a cleaned documentation instance, the filtering process records whether the generated documentation terminates exactly at the matched boundary, which serves as an auxiliary indicator of structural completeness. Only documentation that passes this filtering and rewriting procedure is used as input to *Stage 2*.

C Training Algorithm

We provide the complete training procedure of BatCoder in Algorithm 1.

D Training Settings

The BatCoder models are trained with Reinforce++, implemented on a modified version of the `verl` (Sheng et al., 2024) framework (v0.5.0.dev). Experiments are conducted on four NVIDIA A100 GPUs.

We evaluate BatCoder on two model families: Qwen2.5-Instruct (3B, 7B, and 14B) and Llama-3.1-8B. Different training configurations are adopted for different model scales.

For the Qwen2.5-3B-Instruct model, we set the number of documentation samples to $K = 8$, with a maximum response length of 1500 tokens. The training batch size is 64, the actor learning rate is 1×10^{-6}, and the mini-batch size is 32.

For the larger-scale models, including Qwen2.5-7B-Instruct, Qwen2.5-14B-Instruct, and Llama-3.1-8B, we increase the number of documentation samples to $K = 16$ and use a larger training batch size

Algorithm 1 BatCoder: Self-Supervised Bidirectional Code-Documentation Learning via Back-Translation

Input: Unlabeled code corpus $\mathcal{C}$, model parameters θ, replay buffer $\mathcal{B}$, iterations T, documentation samples K
Initialize θ, $\mathcal{B} \leftarrow \emptyset$
for $t = 1$ to T **do**
  Sample code snippet $c \sim \mathcal{C}$
  Sample documentation candidates $\{d^{(1)}, \ldots, d^{(K)}\} \sim \pi_\theta(d \mid c)$
  for $k = 1$ to K **do**
    Apply filtering and rewriting to $d^{(k)}$, and compute documentation validity reward $R_{\text{doc}}^{(k)}$
    if $d^{(k)}$ fails validity constraints **then**
      Set $R_{\text{code2doc}}^{(k)} \leftarrow 0$
      Store $(c, d^{(k)}, R_{\text{code2doc}}^{(k)})$ in $\mathcal{B}$
      continue
    end if
    Sample reconstructed code $c'^{(k)} \sim \pi_\theta(c' \mid d^{(k)})$
    Compute similarity reward:
      $R_{\text{sim}}^{(k)} = \mathcal{S}(c, c'^{(k)})$
    Set rewards:
      $R_{\text{code2doc}}^{(k)} = R_{\text{sim}}^{(k)} \cdot R_{\text{doc}}^{(k)}$
      $R_{\text{doc2code}}^{(k)} = R_{\text{sim}}^{(k)}$
    Store $(c, d^{(k)}, R_{\text{code2doc}}^{(k)})$ in $\mathcal{B}$
    Store $(d^{(k)}, c'^{(k)}, R_{\text{doc2code}}^{(k)})$ in $\mathcal{B}$
  end for
  Sample minibatches from $\mathcal{B}$
  Update θ using policy gradients for both directions
end for
Output: Optimized model parameters θ

of 128, while reducing the learning rate to 7×10^{-7} to ensure stable optimization at larger scales. The maximum response length and mini-batch size remain unchanged.

We additionally adopt a dynamic sampling strategy (Yu et al., 2025), where minibatches with zero rewards for all trajectories are discarded and excluded from parameter updates.

For ablation experiments with SFT, we use the default configuration, with a batch size of 16, a learning rate of 1×10^{-5}, and a maximum sequence length of 2048.

E Eval Setting and Additional Evaluation on Broader Benchmark Suites

We used `evalplus` framework (Liu et al., 2023) for HumanEval, Humaneval+, MBPP and MBPP+ evaluations, which standardizes test execution and pass@1 computation across benchmarks. We employ greedy decoding (i.e., temperature = 0) with a maximum generation length of 1024 tokens. For MultiPL-E evaluation, we use the `bigcode-evaluation-harness` framework (Ben Allal et al., 2022) with the same decoding settings.

To further assess the generalization ability of BatCoder, we conduct additional evaluations on two broader and more challenging benchmark suites: LiveCodeBench (Jain et al., 2024) and BigCodeBench (Zhuo et al., 2024). These benchmarks complement the primary evaluation in the main text by covering more diverse, realistic, and large-scale coding scenarios beyond canonical function-level tasks such as HumanEval and MBPP.

LiveCodeBench is a large-scale benchmark constructed from recent programming problems, designed to evaluate model performance under realistic and time-sensitive conditions. Compared to HumanEval-style benchmarks, it features more diverse problem distributions and reduces potential data contamination by using temporally filtered datasets. To ensure a fair evaluation, we adopt the LiveCodeBench v5 split starting from October 1, 2024, such that all evaluation instances post-date the release of Qwen2.5. This setup minimizes the risk of test data leakage from pretraining.

BigCodeBench is a comprehensive benchmark that emphasizes more complex and diverse coding tasks, including longer contexts and more intricate problem structures. It provides a broader coverage of real-world programming challenges.

Table 5 reports the performance of BatCoder and base models on these benchmarks.

Model	LiveCodeBench	BigCodeBench
Qwen2.5-3B-Instruct	2.53	40.2
BatCoder-3B	1.88	48.1
Qwen2.5-7B-Instruct	4.31	37.4
BatCoder-7B	4.51	45.9

Table 5: Performance on LiveCodeBench and BigCodeBench.

On BigCodeBench, BatCoder consistently improves over the base models across both scales, demonstrating stronger performance on more complex and diverse coding tasks. On LiveCodeBench, the 7B model shows a modest improvement, while the 3B model exhibits a slight decrease. We hypothesize that this difference is due to the increased difficulty and distributional shift of LiveCodeBench, where smaller models may be more sensitive to generation noise introduced during training.

Overall, these results suggest that BatCoder improves performance on broader and more challenging coding benchmarks, while maintaining competitive generalization under more realistic evaluation

settings.

F Ablation on Similarity Metrics

In this section, we analyze the impact of different similarity metrics used for reward construction in BatCoder. Unless otherwise specified, all ablation experiments are conducted based on Qwen2.5-3B-Instruct. For clarity, we omit the model scale (3B) in the following tables. While the proposed framework is agnostic to the specific choice of metric, different similarity functions may vary in their ability to capture code correctness and functional equivalence. In the main text, BatCoder refers to the variant trained with the CSSG similarity metric.

Impact on Code Generation. We conduct ablation studies by replacing the similarity function in BatCoder with BLEU (Papineni et al., 2002) and CodeBLEU (Ren et al., 2020), while keeping all other training configurations identical to BatCoder-CSSG. Table 6 summarizes the results on HumanEval, Humaneval+, MBPP and MBPP+ benchmarks.

Model	HumanEval(+)	MBPP(+)
Qwen2.5-Instruct	73.8 (68.3)	73.0 (62.2)
BatCoder-BLEU	66.5 (64.0)	71.2 (60.3)
BatCoder-CodeBLEU	75.0 (70.1)	**76.5** (**66.9**)
BatCoder-CSSG	**76.2** (**71.3**)	75.9 (66.4)

Table 6: Impact of different similarity metrics on code generation performance.

As shown in the Table 6, different similarity metrics demonstrate varying performance levels. CodeBLEU and CSSG are similarity metrics specifically designed for code, whereas BLEU is a general text similarity metric. BLEU focuses primarily on lexical overlap and may assign high similarity to code with similar variable or function names despite substantial semantic differences, resulting in less reliable reward signals for code generation. We believe this indicates that as long as the selected metric can accurately reflect code logic and functionality, the BatCoder training framework is effective in enhancing the model's code capabilities.

Impact on Documentation Quality. We further evaluate how different similarity metrics affect the quality of generated documentation. Using the same models, we sample 500 instances from a held-out CodeXGLUE set and generate documentation. The outputs are evaluated by GPT-5.4 across three dimensions: Function Description Clarity, Example Code Correctness, and Readability, each scored on a 0–5 scale. The detailed evaluation prompt is provided in Appendix H.

Model	Clarity	Correctness	Readability	Total
Qwen2.5-Instruct	1.56	0.94	2.27	4.77
BatCoder-BLEU	1.54	0.89	2.26	4.69
BatCoder-CodeBLEU	1.58	0.99	2.31	4.88
BatCoder-CSSG	**1.60**	**1.01**	**2.41**	**5.02**

Table 7: Impact of similarity metrics on documentation quality.

The evaluation results show that while metric-based training can enhance specific aspects of documentation, performance varies by metric. Notably, when using CSSG, BatCoder achieves superior performance in internal logic explanation and structural coherence.

Overall, these results indicate that while BatCoder is compatible with different similarity metrics, choosing a metric that better reflects code semantics leads to stronger performance across both code generation and documentation quality.

G Case Study: Documentation Quality Comparison

To qualitatively analyze the differences before and after BatCoder training, we present a case study comparing the generated documentation for the same code snippet after training with the two approaches. The goal of this analysis is to examine whether BatCoder leads to higher-quality and more informative documentation, even though no explicit reward is imposed directly on documentation content.

The following Ruby helper function is used as the input code for documentation generation:

```
def sign_in(user = create(:user))
  visit new_user_session_path
  within('#new_user') do
    fill_in 'user_email', with: user.
email
    fill_in 'user_password', with: user.
password
  end
  click_button 'Log in'
end
```

This function implements an end-to-end user login procedure in a Rails system test, including

navigation to the login page, form filling, and submission.

Base Model Generated Documentation. The documentation generated by the base model is shown below.

```
require 'capybara'
require 'capybara/dsl'
require 'support/database_cleaner'

# sign_in generates the login process
for a user.
# >>> sign_in(user: create(:user))
# This will log in a user with the
specified email and password.
def sign_in(user = create(:user))
  <end>
```

BatCoder Generated Documentation. In contrast, the documentation produced by BatCoder is as follows:

```
require 'rails_helper'
# The `sign_in` method is used to
simulate a user signing in to the
application.
# It visits the login path, fills in the
 email and password for the given user,
# and submits the form by clicking the
log in button.
# >>> sign_in(create(:user))
# visits the new_user_session_path,
fills in email and password, and clicks
the Log in button
def sign_in(user = create(:user))
  <end>
```

Analysis. Under the filtering and rewriting mechanisms introduced in Appendix B, both outputs satisfy the required documentation format and are therefore considered valid. However, their quality differs substantially.

The base model produces only a coarse and generic description of the function, largely restating that it performs a login operation without capturing the procedural structure or key execution steps. In contrast, BatCoder generates a more detailed and functionally grounded description that explicitly reflects the control flow of the code, including page navigation, form interaction, and submission behavior.

Although BatCoder does not impose an explicit reward on documentation content itself, higher-quality documentation improves the fidelity of the subsequent code reconstruction stage. More precise descriptions enable more accurate back-translation, which in turn yields higher code-level similarity rewards. This indirect supervision effectively guides the model toward generating more informative and semantically aligned documentation, demonstrating the advantage of the proposed bidirectional reinforcement learning framework over the unaligned base model.

H Prompt for Documentation Evaluation

```
Below is a specific code snippet:
{code}

The model was required to generate a
description document for this code. Here
 is the generated documentation:
{doc}

Please evaluate the quality of this
documentation based on the following
three dimensions, using a 0-5 scoring
scale (where 0 is poor and 5 is
excellent).

1. Function Description Clarity: Does
the explanation prioritize underlying
algorithmic flow and mechanisms?
2. Example Code Correctness: Are the
examples syntactically correct and do
they effectively demonstrate the core
logic?
3. Readability: Is the documentation
well-structured and easy to read?

Please strictly output the scores in the
 following format:
<score>Instruction Score,Correctness
Score,Readability Score</score>

e.g.
<score>5,5,5</score>
```

I Future Directions

Several directions are promising for future work.

One direction is to incorporate more diverse reward signals and evaluate them under alternative

reinforcement learning algorithms to assess the robustness of similarity-based rewards.

Another avenue is to further investigate scaling behavior along multiple dimensions, including larger training corpora, increased model capacity, and alternative architectural choices, to better understand the regimes in which BatCoder is most effective.

In addition, the proposed framework may be extended to related tasks such as code completion or code translation.