# Synthetic Semantic Supervision for Contrastive Code Representation Learning in Small Transformers: An Empirical Study

**Kenneth Paulsen**
University of Luxembourg (SnT)
kenneth.paulsen@uni.lu

**Florian Tambon**
University of Luxembourg (SnT)
florian.tambon@uni.lu

**Mike Papadakis**
University of Luxembourg (SnT)
michail.papadakis@uni.lu

**Shin Yoo**
School of Computing, KAIST
shin.yoo@kaist.ac.kr

## Abstract

General-purpose code embeddings power tools for code search, classification, and retrieval. Compact transformer encoders for code typically rely on either human-written docstrings (labor-intensive and inconsistent) or mined structural signals such as execution traces (setting-specific and costly to collect). We empirically study an alternative: contrastive pretraining of small encoders with synthetically generated natural-language descriptions emphasizing code functionality and intent, paired with code in a dual-encoder framework at training and discarded at inference. We benchmark this approach against pretraining-based baselines, generalist LLMs, and embedding-specific models on eight retrieval, classification, and generation tasks across C, C++, and Java. Synthetic semantic supervision yields statistically significant gains over pretraining baselines of the same inference-time size on five of eight tasks, with parity on two more; once fine-tuned, it matches or exceeds zero-shot models two orders of magnitude larger on classification, and it stays on par with execution-aware supervision at matched pretraining data, suggesting a scalable, effective alternative to existing code-representation paradigms.

## 1 Introduction

In recent years, a growing number of automated tools for code search, documentation, analysis, and coding assistance has been proposed (Microsoft; Guo et al., 2022; Huang et al., 2020). These tools rely on effective code representations: abstractions that translate human-written source code into formats suitable for automated processing.[1]

A common approach is to use learned embeddings derived from token sequences or structured program elements (Xie et al., 2023). While

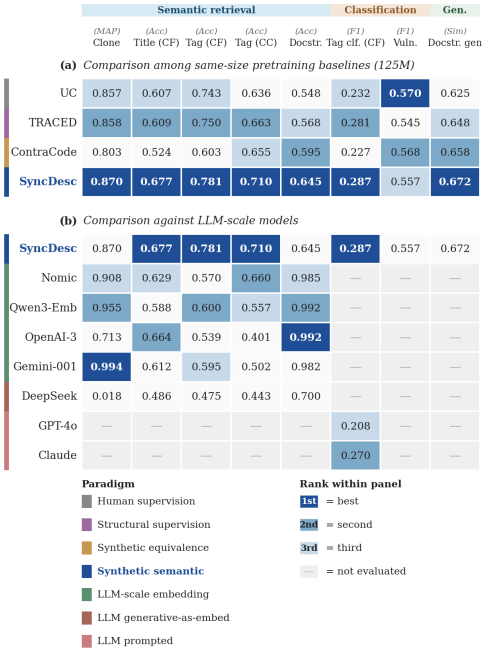


Figure 1: Comparison across eight downstream tasks. Methods grouped by paradigm (left band). Cell color indicates rank within each panel and task column: 1st (dark), 2nd (medium), 3rd (light); cells in columns with a single evaluated method are unshaded; em-dashes indicate the method was not evaluated. **(a)** Against pretraining baselines of the same inference-time size, SyncDesc improves significantly on five of eight tasks and is at statistical parity on two more (paired bootstrap tests in Table 2); ContraCode's ranking carries a cross-language transfer penalty (JavaScript pretraining; Section 3). **(b)** SyncDesc, fine-tuned per task, remains competitive against zero-shot LLM-scale models on algorithmic retrieval and classification. SyncDesc is best on single-source / dual-source variants (per-variant breakdown in Appendix B). Full results in Appendix A.

Large Language Models (LLMs) such as GPT-4o mini (OpenAI, 2024a) can generate such embeddings, and specialized embedding LLMs such as Nomic (Nussbaum et al., 2025) can do so as well, their use incurs high computational and financial costs (e.g., API calls, infrastructure) and scales

[1]Code, data, the generated descriptions, and the generation logs are available at https://zenodo.org/records/22130652.

poorly for many practical applications. This holds in particular for *embedding workloads* (code search, retrieval, and clone detection over large candidate pools), where ranking $N$ candidates requires $N$ precomputed, mutually comparable vectors rather than $N$ prompts per query, and per-query API calls add latency, rate limits, and code egress that private codebases often prohibit. Prompted LLMs retain the complementary advantage of zero-shot generalization; the deployment class we target is the former. In contrast, smaller, specialized transformer models have emerged as a cost-effective alternative, achieving strong performance on targeted coding tasks through fine-tuning (Zeng et al., 2022; Giagnorio et al., 2025; Wang et al., 2025).

Prior work on training code embedding models generally falls into two paradigms. The first, human supervision, uses manually written specifications (e.g., docstrings) as supervision signals to capture program semantics, as in UnixCoder (Guo et al., 2022) or CodeBERT (Feng et al., 2020). While effective, this approach is labor-intensive and struggles to produce representations that generalize across tasks, languages, or evaluation settings (Li et al., 2023). Moreover, existing annotations often rely on surface-level lexical or syntactic cues, providing only indirect access to semantic intent (Panthaplackel et al., 2020), and may contain inconsistencies or errors (Siddiq et al., 2024). The second paradigm, structural supervision, leverages dynamic or control-flow information mined directly from code (e.g., traces, instrumentation, or program analysis), as in TRACED (Ding et al., 2024). While more automated, these approaches can be expensive to collect and must be tailored to specific languages or settings. Recent work has also explored synthetic supervision for code. For example, ContraCode (Jain et al., 2021) proposed generating synthetic equivalent code to pre-train a neural model, while the Qwen2.5-coder white paper (Qwen et al., 2025) demonstrated the use of synthetic data for pretraining code models. These efforts focus primarily on synthetic equivalent code, which limits the diversity of tasks (i.e., they require equivalence), or they train large models, raising the question of the benefit compared to the increase in size. Finally, they focus mainly on general code generation tasks rather than representation learning for downstream understanding tasks.

Despite a wealth of approaches, evaluations and direct comparisons of these approaches remain scarce when it comes to downstream code-oriented tasks. Moreover, while synthetic datasets have proven useful for training LLMs, they remain underexplored for smaller code-oriented transformers. Starting from this observation, we ask: how do existing code-embedding methods fare, and could a simple, scalable approach (using synthetic semantic descriptions and contrastive learning) produce high-quality code embeddings that rival or exceed existing methods? In this paper, we empirically evaluate the following hypothesis: contrastive pretraining with synthetic semantic descriptions can yield code embeddings that (1) outperform traditional pretrained baselines, (2) match or exceed the performance of larger models on downstream tasks, and (3) generalize across languages and tasks, all while being computationally efficient.

To test these hypotheses, we select several baseline approaches from the literature covering a broad spectrum of methods, namely pretraining-based methods (TRACED (Ding et al., 2024), ContraCode (Jain et al., 2021)) applied to a small baseline transformer (UnixCoder (Guo et al., 2022)), large generalist models (GPT-4o (OpenAI, 2024b)), and large generalist embedding models (Nomic (Nussbaum et al., 2025)). We further propose a simple approach, *SyncDesc*, relying on synthetically generated semantic descriptions at scale, paired with code snippets to form aligned supervision for a dual-encoder contrastive objective (Radford et al., 2021), exploiting research highlighting their individual effectiveness (Qwen et al., 2025; Guo et al., 2022). SyncDesc is best understood as lightweight knowledge distillation: it transfers the semantic-extraction ability of a large teacher model into a compact encoder through generated descriptions, rather than introducing a new mechanism. In this respect, none of the compared pretraining paradigms is information-free: TRACED injects privileged execution information, ContraCode injects compiler-verified equivalence, and SyncDesc injects teacher-model semantics; the empirical question this paper answers is which external signal is most effective per unit cost at a fixed inference-time size. Notably, the fine-tuned 125M student exceeds its own teacher on the one task where the teacher is evaluated (Section 3). We further conduct an ablation study that isolates the two ingredients of SyncDesc (the synthetic semantic descriptions and the contrastive objective), comparing against variants supervised with human-written docstrings and trained non-contrastively, to quantify how much each contributes to the observed

gains.

Our results show that contrastively pretrained models with synthetic semantic descriptions improve significantly over pretrained baselines of the same inference-time size on five of eight tasks, with parity on two more and parity-to-below on vulnerability detection (Table 2). On several tasks, SyncDesc, after task-specific fine-tuning that the zero-shot LLMs lack, matches or outperforms larger language models, despite using significantly smaller models. An ablation study confirms that both synthetic descriptions and contrastive learning are critical to these gains, outperforming embeddings supervised by human docstrings or trained non-contrastively. These findings suggest that synthetic semantic supervision, when combined with contrastive learning, offers a scalable, effective, and simple alternative to existing paradigms for code representation learning.

## 2 Experimental Setup

### 2.1 Approaches compared

In our experiments we consider two main families of approaches. First, pre-trained transformers using different pre-training objectives and tasks. For all, we use as a baseline model a **UnixCoder** (Guo et al., 2022) (125M parameters). This model was pre-trained with standard objectives (e.g., masked language modeling / denoising) using human-written docstrings. The first pre-training method is **TRACED** (Ding et al., 2024), an execution-aware baseline trained on CodeNet traces. The second is **ContraCode** (Jain et al., 2021), a pre-training method using synthetic code via equivalent functions. Finally, we add our simple baseline, named in the following **SyncDesc**, relying on contrastive synthetic docstrings, which are detailed in the next paragraph. The second family of approaches is represented by LLMs, either **generalist closed-source LLMs** (GPT-3.5-Turbo (OpenAI, 2023), GPT-4o-mini (OpenAI, 2024a), GPT-4o (OpenAI, 2024b), and Claude 3 variants (Anthropic, 2024)) used as-is, or **embedding LLMs**, i.e. LLMs specifically fine-tuned for embedding generation (Nomic Embed Code (7B) (Nussbaum et al., 2025), Qwen3-Embedding-0.6B (Zhang et al., 2025), OpenAI text-embedding-3-large (OpenAI, 2024c), and Gemini Embedding 001 (Lee et al., 2025)). As a further reference point, we also evaluate a generative code model used directly as an embedder without any embedding-specific training (DeepSeek Coder (6.7B) (Guo et al., 2024)), whose pooled hidden representations we use as-is.

**Proposed baseline** The proposed baseline SyncDesc introduces a simple three-stage pipeline for learning code representations using contrastive learning. First, synthetic natural-language descriptions are generated for each code snippet using a large language model (in our case GPT-4o), ensuring semantic abstraction and avoiding lexical overlap with the source code. Generating synthetic documentation offers several advantages over reusing existing docstrings. Many benchmarks lack proper or consistent documentation (Siddiq et al., 2024), and natural docstrings are often absent, underdeveloped, or weakly aligned with code semantics. Moreover, while using LLMs can introduce errors due to hallucinations, recent studies showed that large language models can generate descriptions that are comparable to, or more semantically consistent than, human-written documentation in some settings (Dvivedi et al., 2024). These code–description pairs are then used to pretrain a dual-encoder architecture with a code encoder (UnixCoder) and a description encoder (a DeBERTa model (He et al., 2020)) with a symmetric contrastive objective, where representations are aligned in a shared embedding space. To improve training, group-aware negative masking excludes pairs from the same problem group from acting as negatives. Finally, after pretraining, only the code encoder is retained and reused for downstream tasks (such as retrieval or fine-tuning) where its fixed-dimensional embeddings serve as input.

### 2.2 Datasets and Tasks

**Pretraining Datasets.** All contrastive models are pretrained using their dedicated process: (i) For TRACED, we use their default C corpus derived from CodeNet (Puri et al., 2021), constructed by following the preprocessing pipeline of TRACED and retaining only *unique static code implementations*. Specifically, multiple execution traces corresponding to the same source code (generated under different inputs) are collapsed into a single code instance, and all dynamic traces are discarded. This results in approximately 18k unique C code samples (24,936 samples; 18,266 unique implementations) used for pretraining (equivalently, the $\sim$121k execution traces reported by (Ding et al., 2024) collapse to $\sim$18k once traces generated from the same source program under different inputs are merged

and dynamic information is discarded, since our setup operates at the level of static code rather than traces), with a disjoint evaluation subset of approximately 1k samples; (ii) for ContraCode, we applied their self-supervised contrastive pretraining pipeline (generating semantically-equivalent program variants through compiler-based source-to-source transformations and training with their instance-discrimination, or MoCo, objective) on the same UnixCoder backbone, using the CodeSearchNet JavaScript corpus ($\sim$1.8M methods) released with their work; this matches the architecture across methods, but the corpus remains JavaScript because ContraCode's augmentation stack is JS-specific and does not port to Java/C without new tooling; the resulting language mismatch is a confound we account for in Section 3, while data volume ($\sim$1.8M vs. $\sim$120k, roughly $15\times$) favors ContraCode; (iii) we use two pre-training datasets for two separate versions of SyncDesc. The first (SyncDesc-1S) only uses a Java corpus aggregating FunCom (Bansal et al., 2023), DeepCom (Hu et al., 2020), and CONCODE (Iyer et al., 2018) method-level examples, yielding a pool of $\sim$200k generated code–description pairs, of which 99,991 Java samples (97,252 unique implementations) form the pretraining set after filtering and generation failures (details in Appendix G); The second (SyncDesc-2S) is a mix of CodeNet C corpora (same as TRACED) and Java samples in equal proportion. Note that this Java corpus is mined from GitHub repositories, not competitive programming; only the C corpus derives from CodeNet.

**Downstream Tasks Datasets.** We evaluate on **eight** downstream tasks spanning retrieval, classification, and generation that were used previously in the literature (Ding et al., 2024; Alon et al., 2018). For small transformer models (i.e., pre-trained with TRACED, ContraCode, and SyncDesc), we fine-tune them using available train datasets. LLMs are used as-is without further fine-tuning. We describe the tasks in the following. **Clone Detection** – *Retrieval setting* (CodeXGLUE-POJ104; (Mou et al., 2016)): Given a code query and a collection of candidate programs, the task retrieves semantically equivalent implementations (same problem) from the candidate set. **Clone Detection** – *Binary classification setting* (BigCloneBench(Svajlenko and Roy, 2015)): Given a pair of code snippets, the task predicts whether the two programs are semantic clones. This dataset also contains human-written docstrings. **Vulnerability Detection** (CodeXGLUE – Defect Detection; (Zhou et al., 2019)): Given a source code function, the task predicts whether it is vulnerable (binary classification). **Docstring Matching** (CodeSearchNet Java; (Husain et al., 2019)): Given a Java function, the task retrieves the corresponding docstring from a candidate pool. **Tags Matching** (CodeChef Java; (Idialu et al., 2024)): Given a Java code snippet, the task retrieves the corresponding algorithmic tag from a candidate pool (**98** tag classes). **Tags and Title Matching** (CodeForces; (Penedo et al., 2025)): Given a competitive programming code snippet, the task retrieves the corresponding tag(s) and title from candidate pools. **Tags Classification** – *CodeForces* ((Penedo et al., 2025)): Given a competitive programming code snippet, the task predicts the tag among the ten most frequent CodeForces tags (multi-class classification). **Tags Classification** – *CodeChef Java* (Idialu et al., 2024): Given a Java code snippet, the task predicts the tag among the ten most frequent CodeChef algorithmic tags (multi-class classification). This dataset also contains human-written docstrings. **Description Generation** (CodeSearchNet Java; (Husain et al., 2019)): Given a Java function, the task generates a natural language description of the code. Four of the eight tasks thus evaluate on GitHub-mined or real-project code (BigCloneBench, Devign, and the two CodeSearchNet-Java tasks).

**Evaluation Metrics.** For each task we follow the evaluation protocol of the original benchmark or prior work: ranking metrics (MAP@K and ranking accuracy) and similarity separation for retrieval (Mou et al., 2016; Husain et al., 2019; Penedo et al., 2025; Idialu et al., 2024), accuracy and F1 for classification (Zhou et al., 2019; Svajlenko and Roy, 2015), and embedding-based semantic similarity against reference docstrings for generation (Husain et al., 2019). Similarity Separation, used as an additional analysis metric, measures the margin between cosine similarity of the query to its positive match versus its negatives; larger values indicate better separation between matched and non-matched pairs and complement ranking accuracy by capturing embedding geometry rather than rank order (Liu et al., 2024).

### 2.3 Hyperparameters and Configuration

**Description Generation.** For our baseline, descriptions are generated with GPT-4o using tem-

perature 0.7, top-$p$ 1.0, and a maximum generation length of 70 tokens. The prompt explicitly instructs the model to produce descriptions of at most 50 tokens, while the higher generation limit serves as a safety margin to prevent truncation. We generated a synthetic docstring for each of the examples in our pretraining datasets. The exact prompt, per-corpus filtering and generation-failure statistics, the full one-time generation cost, and an infrastructure comparison with execution-trace collection are given in Appendix G; the decontamination audit between pretraining corpora and all test sets is given in Appendix F.

**Contrastive Pretraining.** Pretraining in all cases uses AdamW with learning rate $2 \times 10^{-5}$, 10% warmup, mixed precision, and sequence lengths of 512 (code) and 125 (text). SyncDesc-1S is trained for two epochs; SyncDesc-2S models for five epochs. Per-GPU batch size is 3–4 with gradient accumulation of 4.

**Downstream Fine-tuning.** Fine-tuning uses AdamW with learning rate $2 \times 10^{-5}$ and 10% warmup for all methods requiring pre-training (TRACED, ContraCode and SyncDesc). Each experiment is run with three random seeds (42, 123, 456), and results are averaged.

*Data* For fine-tuning the models on the downstream tasks, we used the available train set, except for tasks with over 100K samples. In that case, we sampled uniformly at random and in a way that guarantees a representative sample at a 95% confidence interval. When evaluating, we used only the representative sampled test dataset.

*Loss* We use task-appropriate objectives as a loss function: (i) *contrastive retrieval tasks* (Clone Detection, Docstring Matching, Tags/Title Matching) are fine-tuned with a contrastive objective implemented as cross-entropy over the in-batch similarity matrix between the two task-specific embedding sets (e.g., code–code for clone detection, code–docstring for docstring matching, and code–label text for tags/title matching); (ii) *classification tasks* (Vulnerability Detection, Tags Classification) are fine-tuned with standard cross-entropy loss; and (iii) *generation* (Description Generation) uses token-level cross-entropy with teacher forcing.

*Epochs.* Unless stated otherwise, all downstream tasks are fine-tuned for **1 epoch**. Vulnerability detection is fine-tuned for **5 epochs**, selecting the best checkpoint by validation dataset performance, following the same protocol used in TRACED.

*Generation protocol.* For Description Generation, we keep the code encoder *frozen* and train only the decoder components used for generation, to assess whether pretrained representations support generation without adapting the encoder.

**Hardware.** All experiments run on 4×NVIDIA A100 GPUs (40GB). Seeds are fixed and configurations are logged.

## 3 Results

### 3.1 Comparison to transformer baselines

**Models pretrained with synthetic semantic descriptions improve significantly over the standard transformer baselines and the execution-aware model on five of eight tasks, with statistical parity on two more.** Figure 1 reports all downstream tasks. Gains are largest on semantically oriented retrieval and matching: clone retrieval (MAP 0.870 vs. 0.858), title matching (0.677 vs. 0.609), tag matching (0.781/0.710 vs. 0.750/0.663 on CodeForces/CodeChef), and docstring matching (0.645 vs. 0.595); all four are statistically significant under paired bootstrap tests (Table 2). Tag classification (F1 0.287 vs. 0.281) and description generation (0.672 vs. 0.658) are better on average but not significantly so, and vulnerability detection stays within a narrow band across all models (F1 0.557 vs. 0.570, n.s.; above TRACED at matched pretraining data, 0.557 vs. 0.545), consistent with its reliance on fine-grained control-flow rather than semantic supervision. ContraCode, which trains on synthetically generated equivalent code variants, scores at or below the standard transformer baselines (UC) on most tasks; part of this gap is a corpus confound rather than the supervision paradigm: ContraCode is pretrained on JavaScript (its compiler-based augmentations are JS-specific; Section 2.2) and evaluated cross-language, although architecture is matched (re-executed on the same UnixCoder backbone; the original release used a smaller six-layer transformer) and data volume favors it by roughly $15\times$. Its language-boundedness is a property of equivalence-based supervision, noted descriptively rather than as a performance conclusion. The dual-source variant helps most when the single-source corpus is narrow (C/CodeNet); on Java tasks where the single-source corpus already aggregates three datasets, dual-source provides no consistent gain. A full per-task breakdown and the single- versus dual-source analysis are provided in Appendix B.

**Confound-free comparisons.** For all C/C++ tasks, SyncDesc-S is pretrained on the identical ∼18k CodeNet C corpus used by TRACED, with identical data volume, language, backbone, and fine-tuning protocol, so the supervision signal (synthetic descriptions vs. execution traces) is the only variable. At matched data the two are on par: clone MAP 0.855 vs. 0.858, vulnerability F1 0.557 vs. 0.545, title matching 0.624 vs. 0.609, tag matching 0.743 vs. 0.750, and tag classification 0.220 vs. 0.281 (better on two tasks, worse on one, comparable on two), while requiring no execution infrastructure. The headline gains then come from scaling the descriptions (SyncDesc-2S: +6.8 title, +3.1 tag matching, +0.6 tag F1 over TRACED), a scaling path that trace collection does not offer cheaply (Appendix G). For Java, the cleanest pair is UnixCoder vs. SyncDesc-1S, initialized from it and differing only by the continued contrastive stage: +9.7 docstring matching, +7.4 CodeChef tag matching, +4.7 description generation.

### 3.2 Scale comparison

We compare our approach with large-scale models on both tag classification and retrieval-based tasks. To ensure a fair comparison, we adapt the evaluation protocol to the capabilities of each model family. For classification, we evaluate closed-source large language models (GPT-family and Claude) in a prompted zero-shot setting, explicitly constraining the output space by providing the list of admissible tags and requiring the model to select from this set. For retrieval-based tasks, we evaluate large-scale models in an embedding-based setting by extracting vector representations and selecting the most similar candidates, mirroring the protocol used for our approach. In this setting, we consider different closed/open-source embedding models as well as DeepSeek-Coder, a generalist open source model. We do not include large-scale model comparisons for vulnerability detection and description generation, as discussed above, since these tasks require fine-grained structural or generative supervision that is not the primary focus of our contrastive pretraining objective.

**Classification: Constrained multi-class classification over closely related algorithmic tags benefits more from explicitly learned semantic representations than from generative reasoning alone.** Figure 1 reports classification performance under identical evaluation settings. The best-performing model is **SyncDesc**, which outperforms all prompted large language models, including Claude Sonnet and GPT-4o, despite being over two orders of magnitude smaller (125M parameters) (F1 0.287 vs 0.270). This comparison confounds adaptation capacity with representation quality: SyncDesc is fine-tuned on task-specific data, whereas the prompted LLMs are zero-shot; the result should be read as a fine-tuned 125M encoder matching zero-shot models two orders of magnitude larger, not as superior representation capability. Claude models consistently outperform GPT-family models, but both remain below the dual-source contrastively pretrained encoder, as well as TRACED execution-aware. Prompted LLMs appear less reliable in separating fine-grained algorithmic categories when forced into a closed-label decision space. The gain is nonetheless not mere output distillation of the description generator: the 125M student exceeds its own teacher on this task (F1 0.287 vs. GPT-4o's 0.208), which output imitation alone cannot achieve; swapping the contrastive objective for MLM on identical synthetic data drops 4.2/3.2 points (Table 6).

**Retrieval: Smaller models with dedicated pretraining can perform similarly or outperform large-scale embedding models, doing so at a fraction of the cost.** Figure 1 reports retrieval performance under matched embedding-based evaluation protocols. Across clone detection, title matching, and tag retrieval, SyncDesc is competitive with or outperforms large-scale embedding models. Against DeepSeek Coder, all pretrained methods achieve substantially higher performance, likely because its autoregressive objective is not designed to produce globally comparable embeddings for fine-grained similarity search. Against Nomic Embed Code and the broader set of embedding LLMs (Qwen3-Embedding, OpenAI text-embedding-3-large, Gemini Embedding-001), SyncDesc achieves higher accuracy on title and tag retrieval while trailing on docstring matching, consistent with these LLM embedders being trained primarily on docstring-style data. Notably, SyncDesc achieves this with a model approximately $56\times$ smaller than Nomic Embed Code, highlighting the scale efficiency of semantically enriched contrastive pretraining. This family is matched: as-is usage is the standard regime for embedding models, and all systems share identical

candidate pools and cosine ranking. Table 5 (Appendix B) further removes the fine-tuning asymmetry within the small-model family by evaluating all 125M encoders frozen: on language-matched tasks SyncDesc retains 88–89% of its fine-tuned performance and leads the frozen 125M models; cross-language arms collapse and fine-tuning erases these differences.

### 3.3 Ablation and latent-space analysis

For all ablation and latent space analyses, we restrict evaluation to tasks where human written descriptions are available but are not part of the evaluation objective. This last point means we had to exclude Docstring Matching (CodeSearchNet), as the task objective is to directly evaluate code–docstring alignment with human docstrings. This would bias the ablation study, as the supervision source is the same as the evaluation target. We therefore conduct ablations on Clone Detection and Tag Classification (Java), where description supervision can be varied without affecting the evaluation signal, and additionally on CodeChef Tag Matching, a code-to-text retrieval task whose target texts (algorithmic tag names) are not the supervision source of any variant; this tests text–code alignment directly on a language-matched target (a C++ target would reintroduce the cross-language confound of Section 3).

#### 3.3.1 Ablation Study

**Every ablated variant degrades performance, confirming that both the synthetic descriptions and the contrastive objective are necessary.** We ablate each component of SyncDesc by repeating pretraining and fine-tuning with a single element changed (Figure 2 (a)).

As shown in Figure 2(a), every ablation degrades both Clone Accuracy and Tag F1 relative to SyncDesc (0.724 and 0.276). Semantic content (random text) and substituting MLM for contrastive learning, are the primary drivers of gains. The same ordering extends to retrieval: on CodeChef Tag Matching the full model leads every ablated arm by at least 2.5 points, with human docstrings again last, below random text (Table 7). Full results are reported in Appendix C.

**Supervision-source controls: Semantic extraction from the code, not description writing quality or the specific generator, drives the effect.** On an identical 50k subset of the 1S corpus where only the supervision text varies (Table 8, Appendix C), GPT-4o paraphrases of the human docstrings produced *without access to the code* recover none of the human-to-synthetic gap (clone accuracy 0.728 vs. 0.752 for code-conditioned synthetic; tag F1 0.230 vs. 0.260), and replacing GPT-4o with the open-weights Qwen2.5-Coder-7B as generator matches within seed noise, so the effect is not GPT-4o-specific and generation cost can drop to essentially zero.

Beyond the ablation, we further analyze the properties of the synthetic descriptions themselves. Synthetic descriptions are roughly $3\times$ longer than human docstrings, exhibit substantially higher lexical diversity, and reuse code identifiers far less often (see Appendix D, Table 9). A sensitivity analysis across description lengths (25, 50, 100 tokens) shows no monotonic relationship between length and downstream performance (see Appendix D, Table 10), indicating that abstraction quality matters more than raw length. We return to these properties when interpreting the latent-space analysis below.

**Latent space analysis: Synthetic docstring with contrastive learning improve embedding in small transformer.** To assess how human and synthetic docstrings are represented in the latent space, we measure the average cluster separation (inter-class minus intra-class distance) of their embeddings, using algorithmic category tags (e.g., sorting) and difficulty labels from the CodeChef (Java) test set, and compare against embeddings trained with a frozen description encoder and random docstrings, as in our ablation study. Results in Figure 2 (b, c) confirm that our approach (synthetic docstrings + contrastive pre-training) achieves a higher separation than the frozen encoder on *Problem type* tags (panel (b), 0.292 vs 0.187), and a higher separation than random text on *Problem Difficulty* (panel (c), 0.025 vs 0.012). Human docstrings generally underperform even on *Problem Type*, where the semantics of the tasks should be more evident. In contrast, our approach improves separation, highlighting the stronger semantic embedding, even on *Problem Difficulty*, where semantic alignment might be harder to achieve. The weaker clustering observed for human-written docstrings can be explained by their lexical and structural properties. We show in Appendix D (Table 9) that human docstrings exhibit substantially higher identifier overlap and lower lexical diversity than synthetic descriptions, indicating a tendency to reuse surface-level code tokens rather than abstract

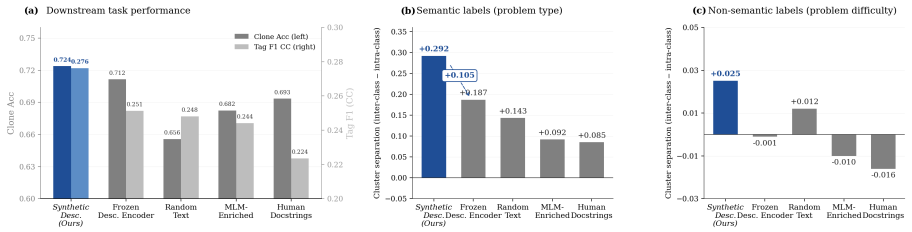


Figure 2: Ablation across alternative supervision sources and training objectives. (a) Downstream-task performance on Clone Accuracy (left axis, dark) and Tag F1 CodeChef (right axis, light); the two metrics use independent y-axis scales to highlight differences within each metric. (b) Cluster separation under semantic algorithmic-category labels. (c) Cluster separation under non-semantic difficulty labels, included as a control. SyncDesc (synthetic descriptions + contrastive objective) achieves the best score across all three panels; alternatives degrade both downstream performance and embedding geometry to varying degrees. Standard deviations and full per-variant numerical results are reported in Appendix C.

functional intent. Moreover, their large variance in length and content leads to inconsistent levels of semantic abstraction across samples. In a contrastive embedding space, such properties reduce the ability to form stable semantic anchors, resulting in blurred cluster boundaries and lower inter-class separation (Chen et al., 2020; Wang and Isola, 2020). In contrast, synthetically generated descriptions enforce consistent abstraction and vocabulary diversity, which promotes tighter grouping of semantically related programs, as reflected in the improved separation scores in Figure 2(b,c). A complementary UMAP visualization is provided in Appendix E.

## 4 Related Work

### 4.1 Models for code embeddings and downstream tasks

Early pretrained code encoders model source code as token sequences using denoising-style objectives. Models such as CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2021) combine masked language modeling with auxiliary signals, including replaced token detection and data-flow modeling, while encoder–decoder architectures like UnixCoder (Guo et al., 2022) and CodeT5 (Wang et al., 2021b) extend this paradigm to support both understanding and generation via multi-objective pretraining. Despite strong performance, these methods achieve only indirect code–text semantic alignment, as supervision relies on reconstruction or generation losses and the quality of naturally occurring comments or documentation. Execution-aware approaches such as TRACED (Ding et al., 2024) incorporate dynamic execution information (e.g., traces, coverage, runtime values), but require executable code, runtime infrastructure, and dataset-specific tooling, which limits scalability. Large autoregressive code LLMs attain strong generative capabilities through massive-scale next-token prediction, as exemplified by GPT-4o-mini (OpenAI, 2024a), while embedding-oriented models like Nomic Embed Code (Nussbaum et al., 2025) train dual encoders contrastively on large code–docstring corpora for retrieval. Although these approaches benefit from scale and data volume, their objectives remain primarily generative or broadly textual and may underperform on specialized tasks. *In contrast*, we propose a synthetic docstring–based contrastive learning approach that improves upon classical semantic alignment as in UnixCoder and scales more effectively than both execution-aware methods and LLMs while achieving competitive performance.

### 4.2 Contrastive Code–Text Alignment

Contrastive learning was first popularized for image–text alignment (Radford et al., 2021) and has since expanded considerably (Hu et al., 2024). In software engineering, it has also emerged as a powerful mechanism for improving code–text alignment, often integrated as a secondary objective in broader pretraining frameworks. For example, UnixCoder and CodeT5+ (Wang et al., 2023) incorporate contrastive objectives alongside masked or generative losses. However, these approaches do not treat contrastive alignment as the primary training signal. Dual-encoder formulations are common in retrieval settings (Husain et al., 2019), but their performance is constrained by the variability and semantic fidelity of developer-written comments or metadata (Sun et al., 2022). Techniques like hard-negative mining (Robinson

et al., 2020) partially address these issues, but alignment quality remains limited by the supervision source. When contrastive alignment is treated as a secondary objective, representation geometry remains largely shaped by reconstruction or generation losses, with contrastive signals acting as a weak regularizer. In contrast, making contrastive alignment the primary training signal directly optimizes the embedding space for semantic similarity and discrimination, which is more aligned with retrieval and matching-based downstream use (Wang and Isola, 2020). Datasets such as FunCom (Bansal et al., 2023), DeepCom (Hu et al., 2020), and CodeSearchNet (Husain et al., 2019) provide large-scale natural-language supervision and underpin prior work in code summarization and retrieval. Their annotations are typically short fragments derived from docstrings or headers and exhibit substantial variability in style, completeness, and abstraction.

A related line of work explores contrastive learning for code through different design axes. ContraCode (Jain et al., 2021), Corder (Bui et al., 2021), SynCoBERT (Wang et al., 2021a), and Concord (Ding et al., 2023) study code-specific augmentations such as variable renaming, dead-code insertion, or other semantics-preserving transformations. SimCSE (Gao et al., 2021) uses dropout-based augmentation for general sentence embeddings. CLEAR (Wei et al., 2022) and CoCoSoDa (Shi et al., 2023) propose retrieval-specific architectures for code search. These works vary the contrastive mechanism (augmentation strategy, dropout, hard negatives) or the architecture. Our contribution is orthogonal: we hold both constant and vary the supervision source, replacing augmentations and human docstrings with synthetic semantic descriptions. The two design dimensions are complementary, and combining synthetic semantic supervision with the augmentation or architectural advances above is a future work direction.

This work treats contrastive alignment as the central objective and pairs code with *synthetic semantic descriptions* designed to provide consistent, instance-level grounding, without relying on potentially missing or low-quality docstrings.

## 5 Conclusion

This study compared code-embedding methods' performance and studied whether a simple approach leveraging synthetically generated semantic descriptions can effectively supervise the learning of general-purpose code representations. The approach, SyncDesc, is a lightweight knowledge distillation: a large teacher's semantic extraction is transferred into a compact encoder and the encoder then serves embedding workloads, effectively keeping the advantage of zero-shot generalization. SyncDesc is evaluated on eight downstream coding tasks (spanning retrieval, classification, and generation) against several transformer baselines across C, C++, and Java, demonstrating better performance on five of the eight tasks. Performance on structurally oriented tasks, such as vulnerability detection, remains comparable to existing baselines, suggesting that this approach complements rather than replaces dynamic supervision. The results indicate that semantically consistent contrastive supervision can produce compact, competitive code representations, matching more complex baselines at matched pretraining data and surpassing them when descriptions are scaled, in semantic tasks while maintaining effectiveness across diverse evaluations. Comparisons with larger models highlight that small scale encoders trained with synthetic semantic supervision can achieve competitive performance, after task-specific fine-tuning that the zero-shot LLMs lack, emphasizing the importance of supervision quality and training objectives, particularly under efficiency and deployability constraints.

Future work could extend synthetic supervision to more programming languages and multi-file or project-level code, analyze how description properties (e.g., abstraction level) interact with downstream tasks, and integrate structured signals (e.g., type information) to enhance representations for tasks requiring deeper semantic or structural reasoning. These directions may further strengthen the potential of synthetic semantic supervision in learning robust, transferable code representations.

## Limitations

Our supervision signal is generated by GPT-4o and may reflect generation artifacts (omitted semantics or stylistic biases) rather than purely generalizable semantic structure. We mitigate this with a fixed prompting template applied uniformly across datasets and languages, but the description quality is not independently verifiable. Exact regeneration also depends on GPT-4o, which may evolve over time.

We evaluate on method-level code across three languages (C, C++, Java), drawn from competitive-programming corpora (CodeNet, CodeForces, CodeChef) and GitHub-mined repository code (FunCom, DeepCom, CONCODE, CodeSearchNet, BigCloneBench, Devign). All training and evaluation nonetheless operate on isolated methods: industrial, multi-file, and project-level distributions (functions whose semantics depend on surrounding modules, frameworks, or build context, and repository-level retrieval) remain untested, as do other languages.

Our 125M-scale baselines are fine-tuned per task, while LLM-scale embedding and prompted models are evaluated as-is, reflecting typical practice but not matched adaptation effort. A frozen-encoder evaluation of all 125M models without any fine-tuning (Table 5) partially removes this asymmetry within the small-model family. We strived to make the comparison across baselines as fair as possible, yet difference in pre-training dataset or objectives might have affected performance on the studied tasks. The ContraCode language confound and the confound-free matched-data comparisons are discussed directly in Section 3.

## Acknowledgments

This work is supported by the Luxembourg National Research Fund (FNR) through the CORE project C23/IS/18182513/MiCE.

## References


Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. Model card and technical report.

Aakash Bansal, Bonita Sharif, and Collin McMillan. 2023. Towards modeling human attention from eye movements for neutral source code summarization. *Proceedings of ACM Human-Computer Interaction, Vol. 7*.

Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '21, pages 511–521. Association for Computing Machinery.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *ICML*.

Yangruibo Ding, Saikat Chakraborty, Luca Buratti, Saurabh Pujar, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. 2023. CONCORD: Clone-aware contrastive learning for source code. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023. Association for Computing Machinery.

Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2024. Traced: Execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12.

Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. 2024. A comparative analysis of large language models for code documentation generation. In *Proceedings of the 1st ACM international conference on AI-powered software*, pages 65–73.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. *Preprint*, arXiv:2002.08155.

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

Alessandro Giagnorio, Alberto Martin-Lopez, and Gabriele Bavota. 2025. Why personalizing deep learning-based code completion tools matters. *ACM Trans. Softw. Eng. Methodol.*, 35(1).

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. *Preprint*, arXiv:2009.08366.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming. *arXiv preprint arXiv:2401.14196*.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2020. Deberta: Decoding-enhanced bert with disentangled attention. *arXiv preprint arXiv:2006.03654*.

Haigen Hu, Xiaoyuan Wang, Yan Zhang, Qi Chen, and Qiu Guan. 2024. A comprehensive survey on contrastive learning. *Neurocomputing*, 610:128645.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25(3):2179–2217.

Qing Huang, An Qiu, Maosheng Zhong, and Yuan Wang. 2020. A code-description representation learning model based on attention. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 447–455.

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*.

Oseremen Joy Idialu, Noble Saji Mathews, Rungroj Maipradit, Joanne M. Atlee, and Meiyappan Nagappan. 2024. Whodunit: Classifying code as human authored or gpt-4 generated – a case study on codechef problems. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*. ACM.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5954–5971.

Jinhyuk Lee, Feiyang Chen, Sahil Dua, Daniel Cer, Madhuri Shanbhogue, Iftekhar Naim, Gustavo Hernández Ábrego, Zhe Li, Kaifeng Chen, Henrique Schechter Vera, Xiaoqi Ren, Shanfeng Zhang, Daniel Salz, Michael Boratko, Jay Han, Blair Chen, Shuo Huang, Vikram Rao, Paul Suganthan, and 43 others. 2025. Gemini embedding: Generalizable embeddings from gemini. *arXiv preprint arXiv:2503.07891*.

Yao Li, Tao Zhang, Xiapu Luo, Haipeng Cai, Sen Fang, and Dawei Yuan. 2023. Do pretrained language models indeed understand software engineering tasks? *IEEE Transactions on Software Engineering*, 49(10):4639–4658.

Lihui Liu, Jinha Kim, and Vidit Bansal. 2024. Can contrastive learning refine embeddings. *arXiv preprint arXiv:2404.08701*.

Philip M. McCarthy and Scott Jarvis. 2010. MTLD, vocd-D, and HD-D: A validation study of sophisticated approaches to lexical diversity assessment. *Behavior Research Methods*, 42(2):381–392.

Leland McInnes, John Healy, and James Melville. 2020. Umap: Uniform manifold approximation and projection for dimension reduction. *Preprint*, arXiv:1802.03426.

Microsoft. [Accessed: 2026-01-06]. [link].

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Zach Nussbaum, John X. Morris, Brandon Duderstadt, and Andriy Mulyar. 2025. Nomic embed: Training a reproducible long context text embedder. *Preprint*, arXiv:2402.01613.

OpenAI. 2023. Gpt-3.5 turbo. https://platform.openai.com/docs/models/gpt-3.5-turbo. Model documentation.

OpenAI. 2024a. Gpt-4o mini: Advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/. Technical overview and release announcement.

OpenAI. 2024b. Gpt-4o system card. System card, OpenAI.

OpenAI. 2024c. New embedding models and API updates. https://openai.com/index/new-embedding-models-and-api-updates/. Accessed: 2026-05-23.

Sheena Panthaplackel, Milos Gligoric, Raymond J Mooney, and Junyi Jessy Li. 2020. Associating natural language comment and source code entities. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8592–8599.

Guilherme Penedo, Anton Lozhkov, Hynek Kydlíček, Loubna Ben Allal, Edward Beeching, Agustín Piqueres Lajarín, Quentin Gallouédec, Nathan Habib, Lewis Tunstall, and Leandro von Werra. 2025. Codeforces. https://huggingface.co/datasets/open-r1/codeforces.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *Preprint*, arXiv:2105.12655.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR.

Joshua Robinson, Ching-Yao Chuang, Suvrit Sra, and Stefanie Jegelka. 2020. Contrastive learning with hard negative samples. *arXiv preprint arXiv:2010.04592*.

Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. CoCoSoDa: Effective contrastive learning for code search. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pages 2198–2210. IEEE Press.

Mohammed Latif Siddiq, Simantika Dristi, Joy Saha, and Joanna CS Santos. 2024. The fault in our stars: Quality assessment of code generation benchmarks. In *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 201–212. IEEE.

Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1609–1620.

Jeffrey Svajlenko and Chanchal K Roy. 2015. Evaluating clone detection tools with bigclonebench. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 131–140. IEEE.

Fali Wang, Zhiwei Zhang, Xianren Zhang, Zongyu Wu, Tzuhao Mo, Qiuhao Lu, Wanjing Wang, Rui Li, Junjie Xu, Xianfeng Tang, and 1 others. 2025. A comprehensive survey of small language models in the era of large language models: Techniques, enhancements, applications, collaboration with llms, and trustworthiness. *ACM Transactions on Intelligent Systems and Technology*, 16(6):1–87.

Tongzhou Wang and Phillip Isola. 2020. Understanding contrastive representation learning through alignment and uniformity on the hypersphere. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021a. SynCoBERT: Syntax-guided multimodal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*.

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.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021b. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 conference on empirical methods in natural language processing*, pages 8696–8708.

Moshi Wei, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, and Song Wang. 2022. CLEAR: Contrastive learning for API recommendation. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 376–387. Association for Computing Machinery.

Yutao Xie, Jiayi Lin, Hande Dong, Lei Zhang, and Zhonghai Wu. 2023. Survey of code search based on deep learning. *ACM Trans. Softw. Eng. Methodol.*, 33(2).

Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pages 39–51.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. 2025. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.

## A Full results with standard deviations

This appendix reports the complete numerical results summarized in Figure 1, including all metrics (e.g., similarity separation) and per-seed standard deviations omitted from the figure for readability. Table 2 additionally reports paired instance-level bootstrap tests for every headline comparison against the strongest 125M baseline: improvements are statistically significant on five of the eight tasks, better on average but not significantly on two (description generation and CodeForces tag classification), and at parity-to-below on vulnerability detection, where SyncDesc-S trails UnixCoder (n.s.) while exceeding TRACED at matched pretraining data (0.557 vs. 0.545).

## B Detailed downstream-task comparison

This appendix expands the condensed comparison in Section 3 with the full per-task breakdown and the single- versus dual-source analysis.

**Comparison to standard baselines and execution-aware pretraining.** Table 1 reports results across all downstream tasks. For single-source, for each downstream task, we used the pre-train dataset matching the programming language of the downstream task. Across retrieval, matching, and classification tasks, models pretrained with synthetic semantic descriptions consistently match or improve upon standard transformer baselines and the execution-aware model, with the improvements statistically significant on five of eight tasks (Table 2). The strongest improvements are observed on semantically oriented retrieval tasks, including clone retrieval (MAP 0.870 for our approach vs 0.858 strongest baseline), title matching (Accuracy 0.677 vs 0.609), tag matching (Accuracy 0.781 vs 0.750 on CodeForces and 0.710 vs 0.663 on CodeChef), and docstring matching (Accuracy 0.645 vs 0.595), where our method attains the highest score across all metrics; all four gains are significant. Improvements are particularly pronounced in similarity separation rather than ranking accuracy, indicating better discrimination between matched and non-matched items. On tag classification, synthetic semantic pretraining is comparable to the strongest baseline on CodeForces (F1 0.287 vs 0.281; not statistically significant). On vulnerability detection, performance differences across models are small and remain within a narrow range, which is consistent with the task's reliance on fine-grained control-flow and implementation-level patterns that are not directly emphasized by semantic-level supervision (F1 0.557 vs 0.570; n.s., and above TRACED at matched pretraining data, 0.557 vs 0.545). For description generation, models pretrained with synthetic semantic supervision achieve the highest semantic similarity scores (Similarity 0.672 vs 0.658 for ContraCode, the strongest baseline; $+1.4$ points, not statistically significant).

**Single-source versus dual-source pretraining.** Table 1 shows that the effect of dual-source pretraining depends strongly on the relative richness of the single-source corpus. For C/C++ tasks (clone retrieval on POJ-104 and CodeForces title, tag matching, and tag classification), single-source pretraining relies exclusively on CodeNet, which provides samples from a single benchmark-style corpus. In this setting, adding Java data (SyncDesc-2S) yields consistent gains over C-only pretraining, e.g., +1.5 MAP on clone retrieval (0.870 vs. 0.855) and +5.7 accuracy points on CodeForces title matching (0.677 vs. 0.624). In contrast, Java-centric tasks are pretrained on a substantially larger and more heterogeneous Java corpus aggregating FunCom, DeepCom, and CONCODE. For these tasks, single-source Java pretraining already provides strong coverage, and adding C data yields no systematic benefit, with performance remaining comparable or slightly lower (e.g., CodeChef tag matching and docstring matching). These results indicate that source mixing primarily compensates for limited single-source coverage, rather than providing uniform gains from increased data volume alone.

**Zero-shot frozen-encoder evaluation.** To assess transfer without any task-specific fine-tuning, Table 5 evaluates all 125M models with frozen, mean-pooled embeddings under the identical protocol used for the embedding-model rows of Table 4 (hence these numbers are not comparable to the fine-tuned results in Table 1). On language-matched tasks, the pretrained encoder transfers directly: SyncDesc-1S-C reaches 0.752 zero-shot clone MAP ($+24.6$ points over frozen UnixCoder; 88% of its own fine-tuned 0.855), and SyncDesc-1S-Java is best among frozen 125M models on CodeChef tag matching (0.632, about 89% of its fine-tuned 0.710, marginally above TRACED's 0.631). We restrict the transfer claim to language-

Table 1: Results for mid-scale (125M) models across all downstream tasks. Metrics are averaged over three seeds with standard deviation. $-S$ = single source, $-2S$ = dual source. For $-S$, for each downstream task, we used the pre-train dataset matching the programming language of the downstream task. We give in **bold** (resp. underlined) the best (resp. second best) approach per task. $^{\dagger}$ marks SyncDesc results whose improvement over the strongest 125M baseline is statistically significant under paired instance-level bootstrap tests (10k resamples, 95% CIs on the metric difference; Table 2).

| Model | Clone | Vuln. | | Title Match (CF) | Tag Classif. (CF) | Tag Match (CF) | | Tag Match (CC) | | Docstr. match | | Docstr. gen |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | MAP ↑ | Acc ↑ | F1 ↑ | Acc ↑ | F1 ↑ | Acc ↑ | Sep ↑ | Acc ↑ | Sep ↑ | Acc ↑ | Sep ↑ | Sim ↑ |
| UnixCoder | 0.857 ± 0.012 | **0.651 ± 0.008** | **0.570 ± 0.007** | 0.607 ± 0.048 | 0.232 ± 0.025 | 0.743 ± 0.005 | 1.397 ± 0.096 | 0.636 ± 0.030 | 1.632 ± 0.096 | 0.548 ± 0.001 | 1.866 ± 0.088 | 0.625 ± 0.055 |
| ContraCode | 0.803 ± 0.006 | 0.632 ± 0.007 | 0.568 ± 0.019 | 0.524 ± 0.005 | 0.227 ± 0.006 | 0.603 ± 0.008 | 1.571 ± 0.111 | 0.655 ± 0.008 | **2.369 ± 0.083** | 0.595 ± 0.031 | 1.378 ± 0.372 | 0.658 ± 0.063 |
| TRACED | 0.858 ± 0.024 | 0.641 ± 0.002 | 0.545 ± 0.002 | 0.609 ± 0.044 | 0.281 ± 0.031 | 0.750 ± 0.012 | 1.395 ± 0.113 | 0.663 ± 0.004 | 1.995 ± 0.047 | 0.568 ± 0.008 | 1.636 ± 0.113 | 0.648 ± 0.004 |
| SyncDesc-S | 0.855 ± 0.009 | 0.646 ± 0.002 | 0.557 ± 0.007 | 0.624 ± 0.062 | 0.220 ± 0.012 | 0.743 ± 0.003 | 1.690 ± 0.201 | **0.710$^{\dagger}$ ± 0.049** | 1.842 ± 0.201 | **0.645$^{\dagger}$ ± 0.011** | 2.283 ± 0.051 | **0.672 ± 0.001** |
| SyncDesc-2S | **0.870$^{\dagger}$ ± 0.014** | 0.6354 ± 0.001 | 0.5225 ± 0.014 | **0.677$^{\dagger}$ ± 0.005** | **0.287 ± 0.029** | **0.781$^{\dagger}$ ± 0.004** | **1.834 ± 0.158** | 0.650 ± 0.045 | 1.547 ± 0.158 | 0.641 ± 0.007 | **2.525 ± 0.056** | 0.670 ± 0.009 |

Table 2: Paired instance-level bootstrap significance of the headline comparisons in Table 1: SyncDesc (per-task headline variant, as labeled) against the strongest 125M baseline on every task, over saved per-item predictions (10k resamples; 95% CI on the metric difference).

| Task | SyncDesc (variant) | Strongest 125M baseline | Δ (pts) | 95% CI | Verdict |
|---|---|---|---|---|---|
| CF Title Match | 0.677 (2S) | 0.609 (TRACED) | $+6.8$ | $[+5.2, +8.4]$ | significant |
| Docstring Match | 0.645 (S) | 0.595 (ContraCode) | $+5.0$ | $[+4.4, +5.6]$ | significant |
| CC Tag Match | 0.710 (S) | 0.663 (TRACED) | $+4.7$ | $[+2.9, +6.5]$ | significant |
| CF Tag Match | 0.781 (2S) | 0.750 (TRACED) | $+3.1$ | $[+1.6, +4.6]$ | significant |
| Desc. Generation | 0.672 (S) | 0.658 (ContraCode) | $+1.4$ | $[-0.4, +3.2]$ | not significant |
| Clone MAP (POJ-104) | 0.870 (2S) | 0.858 (TRACED) | $+1.2$ | $[+0.1, +2.5]$ | significant |
| CF Tag-Clf F1 | 0.287 (2S) | 0.281 (TRACED) | $+0.6$ | $[-1.9, +3.1]$ | not significant |
| Vulnerability F1 | 0.557 (S) | 0.570 (UnixCoder) | $-1.3$ | $[-3.1, +0.5]$ | not significant (below) |

matched settings: cross-language arms collapse on clone retrieval (1S-Java: 0.258), i.e., language match dominates zero-shot embedding geometry, and fine-tuning erases these differences (Table 1). The CodeForces columns are flat across all frozen 125M models, where the large embedding LLMs retain their lead (Table 4). The docstring column is led by vanilla UnixCoder; under this frozen protocol the task is lexically easy (cf. the 0.98+ rows for embedding LLMs in Table 4), and we draw no transfer claim from it.

## C Detailed ablation study

This appendix provides the per-variant ablation analysis summarized visually in Figure 2. For our ablation study, we repeat the pre-train and fine-tuning steps, except that we operate several ablations of our approach to evaluate the contribution towards the performance of each part of our approach. Full numerical results with standard deviations are given in Table 6.

**Human-Docstring** We first repeated the whole process using available human docstrings instead of our generated synthetic docstrings, with the rest of the process unchanged. The goal was to motivate the necessity of synthetic docstrings. This translated into a drop in performance of, on average, approximately 3 percentage points for Accuracy/-Clone detection and approximately 5 percentage points for F1/Tag classification. As such, on top of being scarcely available, native human docstrings generally degrade performance compared to our synthetic docstrings.

**Random-Text** Secondly, we repeated the process by replacing synthetic descriptions with random words of the same length. The goal was to show that performance was not just a byproduct of the training methodology. Swapping in random text degraded performance the most on average: performance dropped by approximately 7 percentage points on Clone detection and approximately 3 percentage points on Tag classification. As such, while fine-tuning certainly helps performance despite random text, the performance of our approach is not driven solely by the training process but also by the added synthetic descriptions.

**Frozen Description Encoder** Thirdly, we repeated the process by freezing the whole description encoder (DeBERTa base + projection layer) when pre-training our approach. The goal was to assess whether the approach did not just benefit from the code encoder learning, and so that the descriptions pairing helped. This translated into a drop of approximately 1 percentage point for Clone detection and approximately 2 percentage points for

Table 3: Code function classification (CodeForces Top-10 Tags). Weighted F1 averaged across three seeds (std. in parentheses). We give in **bold** (resp. underlined) the best (resp. second best) approach per task. Prompted LLMs are evaluated zero-shot with a constrained label set; SyncDesc variants are fine-tuned on task data, so the comparison reflects adaptation regime as well as representation quality (Section 3).

| Model | F1 |
|---|---|
| GPT-4o-mini | 0.151 (± 0.018) |
| GPT-3.5-Turbo | 0.199 (± 0.021) |
| GPT-4o | 0.208 (± 0.017) |
| Claude Haiku | 0.234 (± 0.024) |
| Claude Sonnet | 0.270 (± 0.019) |
| SyncDesc-S | 0.220 (± 0.012) |
| SyncDesc-2S | **0.287 (± 0.029)** |

Table 4: Scale comparison under matched retrieval evaluation protocols. Metrics are averaged over three seeds (std. in parentheses). Superscripts on the SyncDesc row indicate the reporting variant per task ($S$ = single-source, $2S$ = dual-source), matching Table 1. All 125M models are fine-tuned per task; the LLM-scale embedders are used as-is, their standard deployment regime, under identical candidate pools and cosine ranking.

| Model | Clone MAP@499 | Title Match Acc. (CF) | Tag Match Acc. (CF) | Tag Match Acc. (CC) | Docst. match Acc. |
|---|---|---|---|---|---|
| SyncDesc | $0.870^{2S}$ (± 0.014) | $\mathbf{0.677}^{2S}$ (± **0.005**) | $\mathbf{0.781}^{2S}$ (± **0.004**) | $\mathbf{0.710}^{S}$ (± **0.049**) | $0.645^{S}$ (± 0.011) |
| Nomic Embed Code (7B) | 0.908 (± 0.008) | 0.629 (± 0.012) | 0.570 (± 0.015) | 0.660 (± 0.021) | 0.985 (± 0.003) |
| DeepSeek Coder (6.7B) | 0.018 (± 0.009) | 0.486 (± 0.018) | 0.475 (± 0.023) | 0.443 (± 0.027) | 0.700 (± 0.011) |
| Qwen3-Embedding (0.6B) | 0.955 (± 0.001) | 0.588 (± 0.001) | 0.600 (± 0.009) | 0.557 (± 0.002) | **0.992 (± 0.001)** |
| OpenAI text-embedding-3-large | 0.713 (± 0.002) | 0.664 (± 0.029) | 0.539 (± 0.048) | 0.401 (± 0.007) | 0.992 (± 0.001) |
| Gemini embedding-001 | **0.994 (± 0.000)** | 0.612 (± 0.004) | 0.595 (± 0.010) | 0.502 (± 0.003) | 0.982 (± 0.001) |

Tag classification. In practice, this means that keeping the description encoder frozen appears to have limited impact on tasks, meaning the embedding alignment is compensated mainly by the code encoder. Nonetheless, it still decreases performance on average, at a negligible cost since updating the whole description encoder is cheap given its size.

**MLM-Enriched** Finally, we repeated the process by training our approach with masked language modelling (similarly to classical CodeBERT) instead of contrastive learning. The goal was to show that the contrastive objective benefited the approach over the simple masked language modelling task. This resulted in a drop of 4.2 percentage points for Clone detection and 3.2 percentage points for Tag classification. As such, contrastive learning, which uses negative anchoring to train the embedding, seems to improve performance.

**Ablation on a direct text–code alignment target.** The two ablation tasks above are classification tasks; to test the central alignment hypothesis directly, Table 7 repeats the full five-variant ablation on CodeChef Tag Matching, a code-to-text retrieval task whose target texts (algorithmic tag names) are not the supervision source of any variant. We select this task rather than CodeForces title matching because the ablation variants are pretrained on the Java corpus and CodeChef is Java: a C++ target would entangle the supervision-source effect with a cross-language confound. The full model exceeds every ablated arm by at least 2.5 points on a target that no variant was supervised on. Human docstrings again rank last, below even random text, extending the ordering of Table 6 from classification to retrieval; the human-docstring arm also sits close to the plain UnixCoder reference, i.e., human docstrings add little over no continued pretraining here. Because the full model's seed-level standard deviation on this task is large ($\pm 0.049$), we pair this comparison with the instance-level bootstrap protocol of Table 2 (1,890 test items), alongside the similarity-separation geometry already reported in Table 1 (1.842 vs. 1.632 for UnixCoder). We also note that the main results already contain non-docstring text–code alignment evidence: title matching ($+6.8$ points over the best baseline) and tag matching ($+3.1$/$+4.7$ points), where the candidate pools are natural-language titles and tag names.

**Controlled supervision-source suite (paraphrase control and open-weights generator).** To isolate description *writing quality* from *code-derived semantic content*, and to test sensitivity to the generating model, we run four supervision arms on an identical 50k subset of the 1S corpus, differing only in the supervision text, with the identical pretraining recipe and fine-tune/evaluation protocol as Table 6 (3 seeds). Because this suite uses a

Table 5: Zero-shot frozen-encoder retrieval. Mean-pooled embeddings, no fine-tuning; identical protocol to the embedding-model rows of Table 4. Clone and CodeChef tag matching use the full test sets (deterministic); the docstring and CodeForces columns use 10k samples over 3 seeds.

| Model (125M, frozen) | Clone MAP | Docstr. Match | CF Title | CF Tags | CC Tags |
|---|---|---|---|---|---|
| UnixCoder | 0.506 | **0.945** ±0.003 | 0.524 ±0.011 | 0.531 ±0.009 | 0.613 |
| TRACED | 0.346 | 0.877 ±0.004 | 0.511 ±0.003 | 0.514 ±0.007 | 0.631 |
| ContraCode | 0.345 | 0.907 ±0.002 | 0.527 ±0.007 | 0.495 ±0.003 | 0.498 |
| SyncDesc-1S-Java | 0.258 | 0.915 ±0.001 | 0.514 ±0.008 | 0.515 ±0.007 | **0.632** |
| SyncDesc-1S-C | **0.752** | 0.918 ±0.000 | **0.534** ±0.008 | **0.534** ±0.008 | 0.469 |
| SyncDesc-2S | 0.561 | 0.886 ±0.001 | 0.531 ±0.004 | 0.509 ±0.007 | 0.446 |

Table 6: Ablation results across alternative supervision sources and training objectives.

| Variant | Clone Acc | Tag F1 (CC) |
|---|---|---|
| SyncDesc-S | **0.7239 ± 0.016** | **0.2762 ± 0.009** |
| Human Docstrings | 0.6934 ± 0.013 | 0.2235 ± 0.098 |
| Frozen Description Encoder | 0.7116 ± 0.014 | 0.2513 ± 0.052 |
| MLM-Enriched | 0.6824 ± 0.014 | 0.2441 ± 0.051 |
| Random Text | 0.6558 ± 0.011 | 0.2481 ± 0.026 |

Table 7: Five-variant ablation on CodeChef Tag Matching (accuracy, 3 seeds, identical fine-tuning protocol). UnixCoder is shown as a reference, not an ablation arm.

| Variant | CC Tag Match Acc |
|---|---|
| SyncDesc-S (synthetic + contrastive) | **0.710 ± 0.049** |
| Frozen Description Encoder | 0.685 ± 0.030 |
| Random Text | 0.672 ± 0.025 |
| MLM-Enriched | 0.668 ± 0.030 |
| Human Docstrings | 0.655 ± 0.035 |
| UnixCoder (reference) | 0.636 ± 0.030 |

| Supervision source | BCB Clone Acc | CC Tag F1 |
|---|---|---|
| (1) Human docstrings, as-is | 0.739 ± 0.010 | 0.229 ± 0.061 |
| (2) GPT-4o paraphrase of the human docstring (no code) | 0.728 ± 0.006 | 0.230 ± 0.085 |
| (3) GPT-4o synthetic, from code | **0.752 ± 0.003** | **0.260 ± 0.016** |
| (4) Qwen2.5-Coder-7B synthetic, from code | 0.749 ± 0.010 | 0.256 ± 0.025 |

Table 8: Controlled 50k supervision-source suite. Identical 50k code subset of the 1S corpus; arms differ only in the supervision text; identical pretraining recipe and fine-tune/eval protocol as Table 6; 3 seeds.

50k subset, its absolute numbers are not comparable to the full-corpus results in Table 6. Arm (2) has GPT-4o rewrite the human docstrings *without access to the code*, normalizing style, length, and lexical diversity under a no-added-information constraint; arm (4) replicates description generation with the open-weights Qwen2.5-Coder-7B using the paper's prompt and decoding parameters. Results are in Table 8. Rewritten docstrings do not close the gap: clone accuracy 0.728 vs. 0.752 for code-conditioned synthetic (raw human: 0.739), tag F1 0.230 vs. 0.260 (raw human: 0.229); none of the human-to-synthetic gap is recovered by surface quality alone. Both code-conditioned arms sit above both docstring-derived arms on both tasks, with markedly tighter seed variance: what drives the effect is semantic extraction from the code itself, not surface quality. The Qwen2.5-Coder-7B arm matches GPT-4o within seed noise on both tasks (differences of 0.003 and 0.004), so the effect is not GPT-4o-specific and the one-time generation cost can drop to essentially zero on a local GPU with open weights.

## D Description properties

**Lexical and Syntactic properties** To analyze the differences between synthetic and human docstrings and interpret ablation results, the study employs three metrics: average token length, lexical diversity (McCarthy and Jarvis, 2010) (measured via Shannon entropy, the higher the entropy the more diverse the prompt), and identifier overlap (Panthaplackel et al., 2020) (the percentage of code identifiers reused in the docstring, the higher the overlap, the more terms from the code are reused in the prompt). Results are given in Table 9. Synthetic descriptions are, on average, three times longer than human docstrings, which also exhibit three times greater length variability, suggesting inconsistent quality in human-written docstrings. Beyond their brevity, human docstrings show lower lexical diversity, meaning they convey less information per token. Additionally, human docstrings have a higher identifier overlap, indicating a tendency to focus on low-level details (e.g., variable names) rather than high-level functionality. In contrast, synthetic descriptions prioritize general code understanding.

Table 9: Lexical properties of human-written and synthetic descriptions. Values are reported as mean ± standard deviation.

| Source | Avg. Tokens | Lexical Diversity | Identifier Overlap |
|---|---|---|---|
| Human Docstrings | $13.35 \pm 18.24$ | $3.05 \pm 0.98$ | $23.36\% \pm 19.91\%$ |
| Synthetic Descriptions | $48.65 \pm 5.66$ | $5.54 \pm 0.18$ | $5.25\% \pm 5.47\%$ |

Table 10: Sensitivity to description length. Metrics are reported on representative downstream tasks.

| Variant | Clone Acc | Tag F1 (CC) |
|---|---|---|
| SyncDesc (25 tokens) | $0.7143 \pm 0.020$ | $0.2410 \pm 0.0164$ |
| SyncDesc (50 tokens) | $0.7239 \pm 0.016$ | $\mathbf{0.2762 \pm 0.0087}$ |
| SyncDesc (100 tokens) | $\mathbf{0.7248 \pm 0.015}$ | $0.2505 \pm 0.0299$ |

**Synthetic docstring sensitivity to length** We then repeated the pre-training and fine-tuning process , generating new synthetic docstrings with different fixed maximum lengths. The goal is to see whether the chosen synthetic docstring length has any meaningful impact on the embedding, and on the performance of the approach. Results are given in Table 10. Overall, we observe no monotonic or consistent relationship between description length and downstream performance. While increasing length from 25 to 50 tokens improves tag classification, further increasing to 100 tokens does not yield additional gains and can even degrade performance on some tasks. This indicates that description length alone is not a primary driver of representation quality. Importantly, these results do not imply that longer descriptions are inherently better. When considered alongside the lexical and structural properties of synthetic descriptions analyzed above, this suggests that factors beyond raw length, such as consistency and level of abstraction, are likely to play a role.

### D.1 Example synthetic descriptions vs. human docstrings

To make the aggregate lexical metrics in Table 9 tangible, we show three (code, human docstring, synthetic description) triples drawn from the SyncDesc-1S pretraining corpus, spanning a short, a medium, and a long function across distinct algorithmic categories. In each case the human docstring is terse and reuses code identifiers, whereas the synthetic description is longer, abstracts away the concrete names, and states the functional intent, consistent with the averages in Table 9 (human ≈13 tokens, ≈23% identifier overlap; synthetic ≈48 tokens, ≈5% overlap).

```
public double addAndEvaluate(Recommendation<U,I>
    ↪ recommendation){
  double v=metric.evaluate(recommendation);
  if (!ignoreNaN || !Double.isNaN(v)) {
    sum+=v;
    if (!allUsers) {
      numUsers++;
    }
  }
  return v;
}
```

*Human docstring:* "Adds the recommendation metric to the average and returns the user value."
*Synthetic description:* "This routine calculates a score for a given suggestion and updates cumulative statistics. It first computes the score, and if valid under certain conditions, it adds this score to a running total. Additionally, if not all entities are considered, it updates the count of entities. Finally, it returns the computed score for further use or verification."
The human docstring (12 tokens) reuses the identifiers `recommendation` and `metric`, whereas the synthetic description (56 tokens) uses no code identifiers, abstracting them to "suggestion" and "score".

```
@Override public synchronized Extension install(
    ↪ URL url)
    throws InvalidConfigException {
  Preconditions.checkNotNull(url);
  try {
    File tmpFile=download(url);
    Extension extension=loadFromFile(tmpFile);
    finishInstall(tmpFile,extension);
    return extension;
  } catch (InvalidConfigException e) {
    throw e;
  } catch (Exception e) {
    String msg=baseAction.getText(
        "admin.extension.install.error",
        new String[]{url.toString()});
    log.error(msg,e);
    throw new InvalidConfigException(TYPE.
    ↪ INVALID_EXTENSION,msg,e);
  }
}
```

*Human docstring:* "Download and install an extension into local file. The final filename is based on the extension's rowType."
*Synthetic description:* "This routine ensures the download and integration of a software component from a network location. It verifies the input's validity, retrieves the component, and validates its configuration. If errors occur, it logs the issue and raises a specific exception. Upon successful validation, it finalizes the integration process of the component and returns it for further operations."

The human docstring (17 tokens) names `extension` and `rowType`, whereas the synthetic description (56 tokens) summarizes the download-/validate/finalize control flow in domain-neutral terms with negligible identifier overlap.

```
public void reconnectCircuit(){
  ArrayList<Gate> new_Gates=new ArrayList<Gate
    ↪ >();
  ArrayList<Wire> new_Wires=new ArrayList<Wire
    ↪ >();
  if (_Gates != null) {
    for (int i=0; i < _Gates.size(); i++) {
      Gate g=new Gate(_Gates.get(i));
      new_Gates.add(g);
    }
  }
  if (_Wires != null) {
    for (int i=0; i < _Wires.size(); i++) {
      Wire w=new Wire(_Wires.get(i));
      new_Wires.add(w);
    }
  }
  for (int i=0; i < new_Gates.size(); i++) {
    if (_Gates.get(i).Outgoing != null) {
      int index=_Gates.get(i).Outgoing.Index;
      for (Wire w : new_Wires) {
        if (w.Index == index) new_Gates.get(i).
    ↪ Outgoing=w;
      }
    }
  }
  for (int i=0; i < new_Wires.size(); i++) {
    if (_Wires.get(i).From != null) {
      int index=_Wires.get(i).From.Index;
      for (Gate g : new_Gates)
        if (g.Index == index) new_Wires.get(i).
    ↪ From=g;
    }
    if (_Wires.get(i).To != null) {
      int index=_Wires.get(i).To.Index;
      for (Gate g : new_Gates)
        if (g.Index == index) new_Wires.get(i).
    ↪ To=g;
    }
    if (_Wires.get(i).Next != null) {
      int index=_Wires.get(i).Next.Index;
      for (Wire w : new_Wires)
        if (w.Index == index) new_Wires.get(i).
    ↪ Next=w;
    }
  }
  _Gates=new_Gates;
  _Wires=new_Wires;
}
```

*Human docstring:* "Gate and Wire objects must 'point' to each other in memory in a connected circuit."

*Synthetic description:* "This routine reestablishes network connections by duplicating and reassigning components. It replicates nodes and links, then updates relationship references, ensuring each node's outputs and links' endpoints correctly point to the new entities. By iterating over original components, it reconstructs connections in the newly formed structure, maintaining the original network's topology."

The human docstring (15 tokens) leans on the concrete types `Gate` and `Wire`, whereas the synthetic description (50 tokens) abstracts them to "nodes" and "links" and summarizes the topology-preserving reconstruction.

| Pretraining corpus | Test set (items) | Exact | Near ($J \geq 0.7$) |
|---|---|---|---|
| Java 1S (97,252 unique) | Docstring Match (25,246) | 3 (0.01%) | 102 (0.40%) |
| Java 1S | CodeChef Java (1,890) | 0 | 0 |
| Java 1S | BigCloneBench functions (830,832) | 911 (0.11%) | see text |
| C / CodeNet (18,266 unique) | POJ-104 clone (12,000) | 0 | 0 |
| C / CodeNet | CodeForces eval (1,370,195) | 0 | n/a |

Table 11: Decontamination audit: overlap between pretraining corpora and test sets. Exact = MD5 on normalized code; Near = MinHash-LSH near-duplicates.

## E Latent-space (UMAP) analysis

To complement the quantitative cluster-separation analysis in Section 3, we project the learned embeddings using Uniform Manifold Approximation and Projection (UMAP) (McInnes et al., 2020) for 2D visualization. Semantic alignment is evaluated using algorithmic category tags (e.g., sorting) and difficulty labels from the CodeChef (Java) test set. Figure 3 compares UMAP projections of embeddings trained with synthetic descriptions and human-written docstrings. Visually, embeddings from synthetic descriptions exhibit tighter clustering for algorithmic types, suggesting better semantic alignment, while difficulty labels show no clear clustering, likely because difficulty does not directly correlate with semantic content.

## F Decontamination audit

We audit overlap between both pretraining corpora and every downstream test set. Exact duplicates are detected via MD5 hashes of whitespace-stripped, lowercased code; near-duplicates via MinHash-LSH (128 permutations, 5-gram token shingles, Jaccard $\geq 0.7$). Priority was given to the GitHub-Java channel (FunCom/DeepCom/CONCODE against the CodeSearchNet-Java test set and BigCloneBench), where overlap risk is most plausible since pretraining and evaluation draw on overlapping repository ecosystems. Table 11 reports the counts.

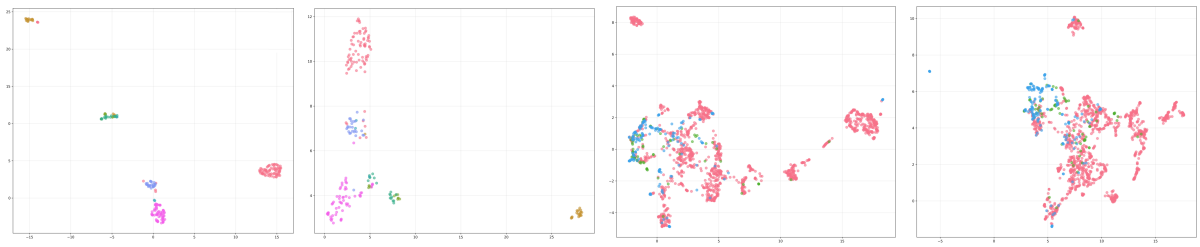

Figure 3: UMAP projections of code embeddings trained with *synthetic descriptions* (first and third panels) and *human docstrings* (second and fourth panels). Colors indicate semantic algorithmic labels (first two panels) and non-semantic difficulty labels (last two panels).

For BigCloneBench, our MinHash configuration proved unreliable on very short functions, so we report the verified exact-hash count (911 functions, 0.11%; the affected test-*pair* fraction is lower, since a pair is affected only if it contains an overlapping function). Given these magnitudes, the expected metric shift on affected metrics is under 0.5%; all C-side test sets have zero overlap.

On generator-side leakage: description generation was conditioned only on pretraining-corpus code, and no test items were ever sent to the GPT-4o API, so the synthetic supervision cannot encode test-set content. In preparing these exact statistics, we found that the originally stated "approximately 200k Java training samples" described the aggregated description pool rather than the pretrained subset; the pretraining sets contain exactly 99,991 Java samples (97,252 unique implementations) and 24,936 C samples (18,266 unique), as now stated in Section 2.2. This strengthens rather than weakens our comparisons: the same results are obtained from roughly half the previously stated data volume, and the data-volume gap to Contra-Code widens to $\sim$15$\times$.

## G Description generation: prompt, statistics, and cost

**Prompt and decoding.** Descriptions are generated with GPT-4o (temperature 0.7, top-$p$ 1.0, maximum 70 generated tokens, with the prompt instructing at most 50 tokens; Section 2). The prompt wrapper adds 180 tokens per call. The exact prompt and decoding scripts are provided verbatim in the replication package.

**Filtering and failure statistics.** The aggregated Java description pool comprised $\sim$200k generated code–description pairs; after filtering and generation failures, the pretraining set contains 99,991 samples (97,252 unique implementations). The C corpus contains 24,936 samples (18,266 unique implementations). The complete generation logs, with per-corpus filtering and failure counts, are shipped with the replication package.

| Scope | Samples | Input tok. | Output tok. | Cost (Batch API) |
|---|---|---|---|---|
| 1S Java corpus | 99,991 | 24.3M | 7.82M | USD 70 |
| C corpus (CodeNet) | 24,936 | 9.4M | 1.96M | USD 22 |
| 2S total | 124,927 | 33.7M | 9.78M | USD 91 |

Table 12: One-time supervision-generation cost, as run.

**One-time supervision-generation cost.** Generation ran via the GPT-4o Batch API (USD 1.25 per million input tokens, USD 5 per million output tokens; rates verified 2026-07-11, unchanged since August 2024 and therefore applicable at generation time). Table 12 gives the full accounting: USD 91 of one-time generation in total, followed by 8 A100-hours of contrastive pretraining. Per-task fine-tuning cost is one epoch on a data subset, and the pretrained encoder is reused across all tasks. As shown in Table 8, an open-weights generator (Qwen2.5-Coder-7B) matches GPT-4o within seed noise, so the generation cost can drop to essentially zero on a local GPU.

**Comparison with execution-trace collection (TRACED).** TRACED's analogous one-time cost is execution infrastructure: compilable programs paired with valid inputs, per-sample multi-input runs, and a language-specific gcc/gdb tracing toolchain. Rerunning their collection pipeline on the same CodeNet C corpus required about 4,020 sequential CPU-hours; the tracer, not compilation, is the bottleneck, since gdb single-steps each program with a per-line variable snapshot. This supervision is also language-bound (the toolchain must be rebuilt per language), whereas description gen-

eration applies uniformly across languages. Zero-shot LLM usage, by contrast, incurs API calls for every task and query, and raises privacy issues for private code.

**Inference throughput.** After pretraining, inference runs at roughly 204 snippets/s even unbatched on a single consumer GPU (RTX 4090, fp16, sequence length 512): embedding the 5,000 POJ-104 test snippets takes 24.5 s under the identical embedding pipeline used for Table 4, one forward pass per snippet (a conservative lower bound of over 17M snippets/day), with no per-query API cost, latency, or rate limits. It also means no code egress, which matters for private codebases where per-query API calls are often prohibited but a one-time generation pass over a vetted corpus is permissible.