Invariant Pretraining for Robust Code Representations

Yifeng He
University of California at Davis
Davis, USA
yfhe@ucdavis.edu

Yundi Xu
University of California at Davis
Davis, USA
xydxu@ucdavis.edu

Christopher Castro Gaw Gonzalo
University of California at Davis
Davis, USA
ccgawgonzalo@ucdavis.edu

Zili Wang
University of California at Davis
Davis, USA
zliwang@ucdavis.edu

Hao Chen
The University of Hong Kong
Hong Kong
chenho@hku.hk

Abstract

Encoder-based code representation models remain widely used for discriminative tasks such as clone detection and code classification because their small size and low inference cost matter. Yet on *invariant programs*, semantically equivalent code with different syntax, their representations degrade substantially despite unchanged behavior. We measure this gap across four encoder baselines, two tasks, and four datasets, then test a minimal code-only continued pretraining recipe that consistently recovers part of it. Invariant pretraining (InvPT) applies semantics-preserving transformations and combines masked language modeling with multi-positive supervised contrastive learning. All augmentations of a source function are positives: self-contrast pairs use the same code with different masks, whereas invariant-contrast pairs use transformed code, providing varied difficulty without paired natural-language data. Across all model–dataset comparisons, InvPT improves robustness on transformed test sets by a median of 8.1 percentage points for clone detection (up to 11.0) and 3.6 points for code classification (up to 19.2), while matching or improving standard performance. Ablations identify multi-positive invariant contrast as the main source of these gains. Because the test transformations are composed from the pretraining operator family, the results establish invariance only to that family, not robustness in general. Thus, the contribution is not a new objective but a measurement of where encoder robustness breaks and how much of it a simple code-only recipe recovers.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Computing methodologies → Neural networks**; *Unsupervised learning*.

LMPL '26, Oakland, CA, USA

ACM ISBN 979-8-4007-2986-7/2026/10
https://doi.org/10.1145/3843750.3843840

Keywords: Code Representation Learning, Model Robustness

ACM Reference Format:
Yifeng He, Yundi Xu, Christopher Castro Gaw Gonzalo, Zili Wang, and Hao Chen. 2026. Invariant Pretraining for Robust Code Representations. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Language Models and Programming Languages (LMPL '26), October 4–9, 2026, Oakland, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3843750.3843840

1 Introduction

Machine learning for code is now dominated by large generative models [1, 2, 3, 4]. Encoder-based code representation models nonetheless remain the practical choice for discriminative tasks such as clone detection and code classification, which still sit inside modern pipelines: deduplicating training corpora, retrieving and ranking code, flagging near-duplicates in agent workflows. Models such as CodeBERT [5] and GraphCodeBERT [6] are far smaller than generation models while performing well on these tasks. With roughly 125M parameters, they can match or outperform generation models in the 7B–15B range while requiring less storage and lower inference cost [7, 8]. Their low fine-tuning cost keeps them attractive wherever robust code representations are needed at scale, so their failure modes still matter.

That robustness is not assured. Modern programming languages are syntactically flexible: developers routinely express the same computation in different ways because of coding style, conventions, or language idioms. We refer to such semantically equivalent but syntactically varied programs as *invariant programs* (or simply *invariants*), because their observable behavior is unchanged under these rewritings. For example, a counting `for` loop can often be rewritten as an equivalent `while` loop without changing program behavior. These variations are common in practice, yet code representation models often mishandle them because pretraining encourages sensitivity to surface tokens [9]. Ahmed and Devanbu [10] showed that perturbing natural-language elements in code can substantially degrade model performance. How large this degradation is across current encoders, and

how much of it a simple training change can remove, has not been measured systematically.

Prior work addresses parts of the problem but does not settle it. Yu et al. [11] introduced SPAT, a semantics-preserving transformation-based data-augmentation method that improved downstream performance without explicitly targeting robustness. Wang et al. [12] combined SPAT with curriculum learning, but focused on fine-tuning rather than pretraining. Liu et al. [13] proposed contrastive pretraining, but relied on transformations that can alter semantics or introduce syntax errors, and on *bi-modal* corpora with paired natural language and code (NL-PL pairs). From CodeBERT [5] to ContraBERT [13], these methods train on function–docstring pairs, and ContraBERT further designs natural-language transformations for its contrastive objective. For robustness against invariant programs, however, paired NL-PL data is not essential: the supervisory signal already lives in code through semantics-preserving transformations.

We contribute an empirical study, not a new objective. We quantify the robustness gap of encoder code models under invariant programs and ask how far a deliberately minimal, code-only recipe can close it. The recipe, invariant pretraining (InvPT), is continued pretraining that uses only programming-language data. InvPT applies semantics-preserving transformations to the pretraining corpus (Section 2.1), optimizes encoders with supervised contrastive learning (Section 2.2), and mixes self-contrast with invariant-contrast pairs to provide positives of varying difficulty (Section 2.2.3). Its only non-obvious design choice is a multi-positive contrastive mask that treats all augmentations of the same source function as positives, avoiding the false-negative problem of standard InfoNCE, which would push semantically equivalent variants apart. This code-only design removes the need for paired NL-PL corpora and enables continued pretraining on large PL-only code collections. The resulting models gain robustness on code-to-code tasks while preserving or improving standard downstream performance.

In summary, our contributions are:

- **(i)** We measure the robustness of four encoder code models to semantics-preserving transformations across clone detection and code classification on POJ104 and three CodeNet subsets (Java250, Python800, C++1400), quantifying a consistent degradation under semantically equivalent rewrites.
- **(ii)** We instantiate InvPT, a code-only continued pretraining recipe that combines invariant transformations with a multi-positive supervised contrastive objective. It gains up to 6.57 points of MAP@R on the original clone-detection test sets and improves robustness on every transformed model–dataset comparison, by a median of 8.07 points on clone detection and 3.56 on classification, recovering part but not most of the degradation.
- **(iii)** We run ablations isolating the source of improvement, showing that multi-positive invariant contrastive learning is the dominant factor, that self-contrast adds consistent gains, and that paired natural-language descriptions are unnecessary for invariant pretraining.

2 Invariant Pretraining

Figure 1 shows InvPT's three components: invariant code transformations, code-only contrastive learning, and mixed-difficulty contrastive pairs. For each source snippet, we generate semantically equivalent but syntactically varied variants, and continue pretraining on both the original and its variants with a masked language modeling objective combined with an invariant contrastive objective. Self-contrast and invariant-contrast examples are mixed so the model sees positives of varying difficulty during training.

2.1 Invariant Code Transformations

Invariant programs—semantically equivalent code that differs syntactically—arise naturally from coding styles, conventions, and language idioms [14], yet models often mishandle them because pretraining memorizes surface form [12, 13, 15]. We address this by pretraining with invariant code transformations that produce semantically equivalent but syntactically diverse variants. Following previous work on naturally occurring invariants [16, 11], we design six transformation operators in two categories: *syntax* operators that modify surface-level tokens and a *branching* operator that alters control flow while preserving runtime behavior. Table 1 lists each operator with its transformation rule and supported languages, where p denotes a predicate (condition) and B a code block. We describe each operator below with its preconditions and a correctness argument.

We implement all operators at the AST level to ensure syntactic correctness: we reuse the Java operators of Yu et al. [11], use `clang` for C/C++, and the Python `3 ast` module for Python (with `2to3` as a fallback for Python 2 code [17]). We discard code snippets with syntax errors that prevent AST parsing. Not all operators apply to every language (*e.g.*, Python's iterator-based for-loops preclude `W2F` and `F2W`), as shown in Table 1. Our operators target the language constructs that appear in our benchmarks (POJ104 and CodeNet); for Python we therefore omit the loop and increment operators.

Variable Rename (`VarRe`). Programmers choose variable names freely without affecting runtime behavior; a name may be a whole word, an abbreviation, a single character, a term in another language, or a random string. `VarRe` replaces all variable names with randomly generated, non-repeating strings that comply with the naming conventions of the target language, keeping the transformed code syntactically valid. Variable names are purely syntactic identifiers with no effect on runtime behavior in any supported language (Java,

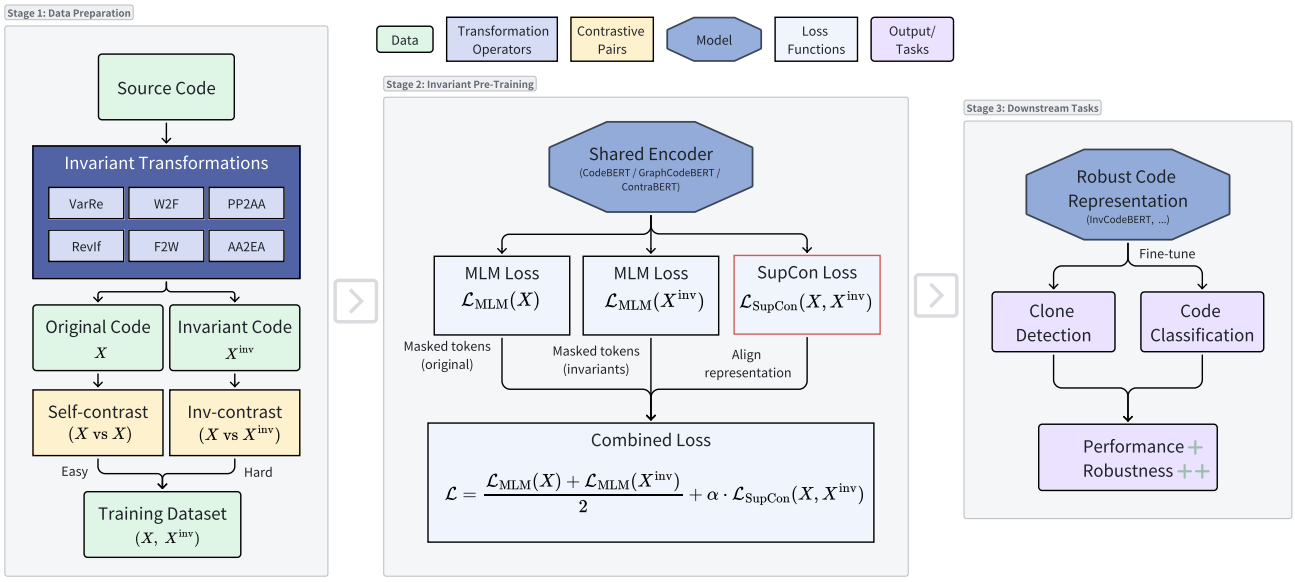

Figure 1. Overview of invariant pretraining (InvPT).

Table 1. Invariant transformation operators, transformation rules, and supported languages.

Operator	Category	Transformation rule	Lang.
`VarRe`	Syntax	Replace all variable names with random valid identifiers	C/C++
`W2F`	Syntax	`while` (p) $\{B\}$ $\Rightarrow$ `for` $(;p;)$ $\{B\}$	C/C++
`F2W`	Syntax	`for` $(init; p; inc)$ $\{B\}$ $\Rightarrow$ $init$; `while` (p) $\{B; inc\}$	C/C++
`PP2AA`	Syntax	`x++` $\Rightarrow$ `x += 1`	C/C++
`AA2EA`	Syntax	`x += 1` $\Rightarrow$ `x = x + 1`	C/C++
`RevIf`	Branching	`if` (p) $\{B_1\}$ `else` $\{B_2\}$ $\Rightarrow$ `if` $(\neg p)$ $\{B_2\}$ `else` $\{B_1\}$	C/C++

C/C++, Python), and because `VarRe` performs a consistent, injective renaming of all occurrences within each scope, data and control flow are unchanged. Naming variation is pervasive in real code: style guides differ, internationalization yields non-English identifiers, and abbreviation conventions vary by domain [14], so `VarRe` models a naturally occurring transformation rather than an artificial adversarial perturbation.

***While-to-For* (`W2F`).** While- and for-loops are both common imperative constructs that programmers use interchangeably. Converting a while-loop to a for-loop reuses the condition while leaving the initialization and increment slots empty:

$$\texttt{while}\ (p)\ \{B\} \ \Rightarrow\ \texttt{for}\ (;p;)\ \{B\}.$$

The for-loop `for(; p;){B}` evaluates p before each iteration and executes B if true, identical to `while(p){B}`; the empty slots add no computation, so semantics are preserved exactly.

***For-to-While* (`F2W`).** The reverse operator converts a for-loop into a while-loop, a more involved rewrite because while-loops lack a built-in counter. We hoist the counter initialization before the loop and append the increment at the end of the body:

$$\texttt{for}\ (init; p; inc)\ \{B\} \ \Rightarrow\ init;\ \texttt{while}\ (p)\ \{B; inc\}.$$

`F2W` applies only to for-loops whose body has no `continue` statement: with `continue`, the appended inc would be skipped, whereas the original for-loop runs inc before the next condition check regardless. Our `clang`-based implementation for C/C++ and the Java implementation of Yu et al. [11] both check this precondition. When $init$ introduces a declaration, we wrap the rewrite in a block (*i.e.*, $\{\ init;\ \texttt{while}\ (p)\ \{B; inc\}\ \}$) so loop-local declarations do not leak into the enclosing scope. Under this precondition, $init$ runs once, p is checked before each iteration, B is the body, and inc runs after each complete iteration, matching the for-loop semantics exactly.

PlusPlus-to-AddAssign (PP2AA). Self-increment operators are common and often interchangeable. PP2AA converts an increment `x++` or `++x` into `x += 1` (in this direction only). For post-increment we check on the parent AST node that the result value is not consumed, since `x++` evaluates to the old value whereas `x += 1` evaluates to the new one; pre-increment needs no such precondition. We further restrict the operator to variables of primitive numeric type, so overloaded `operator++` in C++ is never rewritten.

AddAssign-to-EqualAdd (AA2EA). AA2EA converts "add-assign" into "equal-add": `x += 1` $\Rightarrow$ `x = x + 1`. For primitive numeric types, which dominate our benchmarks (POJ104, CodeNet), `x += expr` and `x = x + expr` are equivalent in all supported languages. For C/C++ the `clang`-based implementation enforces this precondition directly, rewriting only variables whose declared type is primitive numeric. Python admits no such static check: the declared type is unavailable at the AST level, so our implementation rewrites every augmented assignment, and for objects that define `__iadd__`, such as lists and `numpy` arrays, `x += e` mutates in place while `x = x + e` rebinds a fresh object, which is observable under aliasing. The rewrite is sound for the primitive-scalar arithmetic that dominates our competitive-programming corpora, but it is a heuristic rather than a guaranteed invariant on arbitrary Python (Section 5).

Reverse If (RevIf). RevIf reverses the condition of an `if` statement. For a single-branch `if` (no `else`), we negate the condition, leave an empty branch, and move the original body to a new `else`:

$$\texttt{if}\ (p)\ \{B\} \ \Rightarrow\ \texttt{if}\ (\neg p)\ \{\}\ \texttt{else}\ \{B\}.$$

For a two-branch `if`, we negate the condition and swap the branches:

$$\texttt{if}\ (p)\ \{B_1\}\ \texttt{else}\ \{B_2\} \ \Rightarrow\ \texttt{if}\ (\neg p)\ \{B_2\}\ \texttt{else}\ \{B_1\}.$$

For `else if` chains (syntactic sugar for nested `if`s), we apply these rules recursively to each nested `if` node in the AST. Negating p and swapping branches preserves the mapping from condition values to blocks; for single-branch `if`s, the empty block has no effect; and recursive application is correct because each nested `if` is reversed independently.

2.2 Invariant Contrastive Learning

Because our semantics-preserving transformations preserve semantics under the preconditions of Section 2.1, the positive-pair gradient signal is unambiguous, and we use a single shared encoder rather than the momentum-decoupled encoder adopted by ContraBERT [13, 18]. We train with the average MLM loss on the original code and its invariant, combined with an invariant contrastive loss. Since the transformations can produce multiple positives per anchor, we use supervised contrastive learning, which accommodates all positives without pushing equivalent augmentations apart as InfoNCE does.

2.2.1 Masked Language Modeling.

Masked language modeling (MLM), a common pretraining objective for language models, trains the model to predict masked tokens in the input sequence [19]. Given an input sequence $X = \{[CLS], C, [EOS]\}$, where C is the tokenized code snippet, we randomly mask 15% of the tokens in X to obtain a masked sequence X', following previous work [6, 13, 19, 20]. Let $M_X \subset X$ denote the masked tokens in X. The loss is the negative log-likelihood of the masked tokens given X':

$$\mathcal{L}_{\text{MLM}}(X) = -\frac{1}{|M_X|} \sum_{x \in M_X} \log P(x|X').$$

2.2.2 Supervised Contrastive Mask.

Because our dataset applies multiple transformation operators to the same source function (*e.g.*, both `VarRe` and `RevIf`), different augmentations of the same function can co-occur in a mini-batch. Under InfoNCE [21], these semantically equivalent augmentations would be *pushed apart* as negatives, contradicting the invariance objective. We therefore adopt supervised contrastive learning (SupCon) [22] with a multi-positive mask: each snippet receives a deterministic identifier computed from its original source code (a truncated SHA-256 hash), so all augmentations of the same function share an identifier and are recognized as positives.

For a mini-batch of B code–invariant pairs, we concatenate the ℓ_2-normalized CLS embeddings into a pool of $N = 2B$ representations $\mathbf{z}$ with duplicated identifiers ids, and form a multi-positive mask $M_{ij} = \mathbb{1}[\text{ids}_i = \text{ids}_j \wedge i \neq j]$. Let $\mathcal{P}(i) = \{j : M_{ij} = 1\}$; every anchor has at least its paired augmentation as a positive, and when the same source function appears k times, each anchor has up to $2k - 1$ positives. We compute the identifier from the canonical, non-augmented source c as

$$\text{id}(c) = \text{int64}\big(\text{SHA-256}(c)[:8]\big)\ \&\ \texttt{0x7FFFFFFFFFFFFFFF},$$

the first 8 bytes of the digest read as a big-endian 64-bit integer with the sign bit masked. Because we hash the canonical source, all augmentations of a function, including self-contrast copies (Section 2.2.3), share an identifier. The assignment is deterministic and needs no coordination across training processes or data shards. Table 2 illustrates the resulting mask for $B = 4$ pairs drawn from three source functions f, g, h, where f appears twice through different operators (indices 0–3 are original codes, 4–7 their augmentations): anchors for f (indices 0, 3, 4, 7) each have 3 positives, whereas anchors for g and h each have exactly 1.

Let τ denote the temperature, and let $\text{sim}(\mathbf{u}, \mathbf{v}) = \mathbf{u}^\top \mathbf{v}/\tau$ be the scaled cosine similarity of ℓ_2-normalized vectors. The

Table 2. Example positive mask M for $B = 4$ with source functions $[f, g, h, f]$. Rows/columns 0–3 are original codes; 4–7 are their augmentations. "•": the positives ($M_{ij} = 1$); "-": the excluded diagonal.

	0	1	2	3	4	5	6	7
0 (f)	-			•	•			•
1 (g)		-				•		
2 (h)			-				•	
3 (f)	•			-	•			•
4 (f)	•			•	-			•
5 (g)		•				-		
6 (h)			•				-	
7 (f)	•			•	•			-

supervised contrastive loss is

$$\mathcal{L}_{\text{SupCon}} = -\frac{1}{|\mathcal{A}|} \sum_{i \in \mathcal{A}} \frac{1}{|\mathcal{P}(i)|} \sum_{p \in \mathcal{P}(i)} \log \frac{\exp\big(\text{sim}(\mathbf{z}_i, \mathbf{z}_p)\big)}{\sum_{j \neq i} \exp\big(\text{sim}(\mathbf{z}_i, \mathbf{z}_j)\big)},$$

where $\mathcal{A} = \{i : |\mathcal{P}(i)| > 0\}$ is the set of anchors with at least one positive. When no same-function collisions occur, the mask reduces to the diagonal pairing of InfoNCE. For a batch of code X and invariants X^{inv}, the overall objective is

$$\mathcal{L}(X, X^{\text{inv}}) = \frac{\mathcal{L}_{\text{MLM}}(X) + \mathcal{L}_{\text{MLM}}(X^{\text{inv}})}{2} + \alpha \mathcal{L}_{\text{SupCon}},$$

where α weights the contrastive loss.

2.2.3 Mixed-Difficulty Contrastive Pairs.

Self-contrast. SimCSE [23] showed that passing the same input through an encoder with different dropout masks produces effective contrastive pairs for sentence representation learning. We introduce *self-contrast* to code representation learning, a signal that prior work [11, 12, 13] has overlooked, perhaps because small code edits can change semantics. In InvPT, we include identical code snippets as contrastive pairs during pretraining, relying on the 15% MLM masking probability to produce varied positive pairs from the same code snippet. Self-contrast provides the easier signal in our mixed-difficulty strategy: the model learns to produce similar embeddings for the same code with different masked tokens.

Invariant-contrast. The single-operator transformations of Section 2.1 supply the hard end of the range. Whereas a self-contrast pair differs only in masked positions, an invariant pair can differ substantially in surface form while preserving semantics. For example, `F2W` rewrites a `for`-loop into a `while`-loop by hoisting the initialization before the loop and appending the increment to the body, and `VarRe` replaces every identifier with a fresh random name, so an anchor and its positive may share almost no tokens. Aligning such a pair forces the model past surface cues toward the underlying computation, a strictly harder objective than tolerating masked tokens on an otherwise identical sequence.

Mixing difficulties. We expose the model to both pair types simultaneously rather than staging them, presenting each anchor with its easy (self-contrast) and hard (invariant) positives in the same batch. A staged schedule that withheld the harder pairs early and introduced them later would shift the positive-pair distribution mid-training, risking instability and catastrophic forgetting [24] during continued pretraining. We further start training with a low learning rate of 2e−5 so the model adapts gradually to the contrastive objective, then control the rate with standard warm-up and scheduling [25]. The ablation in Section 3.4 supports this mix: keeping only the hard invariant pairs and removing self-contrast lowers performance on every original benchmark, so the easy signal complements the hard one rather than duplicating it.

3 Experiments

3.1 Tasks, Models, and Setup

We evaluate InvPT on two code-to-code downstream tasks, where robustness to invariant transformations is particularly important. **Clone detection** retrieves semantically equivalent programs from a candidate pool using cosine similarity over encoder embeddings. We evaluate on POJ104 [26] (104 problems, 500 C/C++ solutions each) and three CodeNet [27] subsets (Java250, Python800, and C++1400) following Zhao et al. [17]. **Code classification** predicts the functional category (problem number) of a program; we evaluate on the same POJ104 and CodeNet benchmarks as clone detection.[1]

We compare CodeBERT [5], GraphCodeBERT [6], ContraBERT_C, and ContraBERT_G [13], together with their InvPT counterparts.[2] We follow the standard data splits and fine-tuning settings used in prior work [35, 27, 17]; Table 3 lists the correspondence between each baseline and its InvPT variant. For clone detection, we report Mean Average Precision at R (MAP@R) [36], and for code classification we report accuracy. Relative improvements (%) over the corresponding baseline appear in gray.

We pretrain on the Java and Python subsets of CodeSearchNet [37]: it provides high-coverage corpora for both languages, and all six transformation operators apply to them. We hold C/C++ out of pretraining on purpose. The cross-language evaluation in Section 3.2 and Section 3.3 then tests

[1] We do not use BigCloneBench [28], due to label quality concerns [29, 30] and heavy data contamination. We also do not use Devign [31] defect detection due to serious data quality and label accuracy concerns [32].
[2] We additionally include CodeSage [33] and ModernBERT [34] as reference points in the clone detection evaluation (Table 4). Since these models use different architectures, we do not apply InvPT continued pretraining to them.

Table 3. Correspondence between baseline models and our InvPT models.

Baseline	CodeBERT	GraphCodeBERT	ContraBERT_C	ContraBERT_G
+ InvPT	InvCodeBERT	InvGraphCodeBERT	InvContraBERT_C	InvContraBERT_G

whether structural invariance learned on Java and Python transfers to a disjoint target language, rather than measuring in-distribution performance. We pretrain with maximum sequence length 512, batch size 256, temperature $\tau = 0.1$, and contrastive weight $\alpha = 1.0$, using AdamW [38] with learning rate 2e−5, weight decay 0.01, and 10% warmup, for 3 epochs with checkpoints selected by validation loss. For downstream fine-tuning, we follow CodeXGLUE [35] for the POJ104 split and Puri et al. [27] for the CodeNet splits, using learning rate 2e−5, batch size 8, and maximum sequence length 400. All pretraining runs use four NVIDIA H100 GPUs with 80 GB memory each, and all downstream runs use a single NVIDIA H100 GPU. Continued pretraining is inexpensive relative to pretraining from scratch: each InvPT model takes about 20 hours on the four H100s, or roughly 80 GPU-hours, for the three epochs over the augmented corpus. All reported numbers come from a single pretraining and fine-tuning run per configuration, with no repeats across random seeds (Section 5).

We organize the evaluation around the claims made in the introduction. Throughout, we use *standard performance* to refer to performance on the original (untransformed) test set, in contrast to robustness, which we measure on the transformed test sets. We first test whether InvPT preserves standard downstream performance. We then evaluate robustness on transformed test sets to measure invariance under unseen compositions of transformations. Next, we use ablations to identify which design components drive the gains. Finally, we present representation visualizations as qualitative evidence that is consistent with the quantitative results.

3.2 Standard Downstream Performance

We first evaluate standard downstream performance to verify that InvPT does not buy robustness at the cost of performance on the original test sets. We then turn to transformed test sets in Section 3.3, where robustness to invariant programs is the main target. Table 4 shows the clone detection results. Applying InvPT improves all four baseline families on most benchmarks, and InvGraphCodeBERT and InvContraBERT_G achieve the strongest overall performance. The largest gain appears on POJ104, where InvGraphCodeBERT improves GraphCodeBERT by 7.8%. Notably, these gains also transfer to the C/C++ benchmarks. Although the base encoders were pretrained on C/C++ (among other languages), our invariant continued pretraining uses *only* Java and Python programs from CodeSearchNet; no C/C++ code receives any invariant transformation during this stage. The C/C++ improvements therefore reflect cross-language transfer of the robustness signal acquired from Java and Python invariants alone.

Code classification accuracy is nearly saturated on these benchmarks (orig columns of Table 5). Even in this regime, InvPT matches or slightly improves the corresponding baselines on all four datasets.

3.3 Robustness Against Invariant Programs

We evaluate robustness by cumulatively applying all six transformations from Section 2.1 to the test set. Because training uses only single-operator transformations, these compositions are unseen, so the setting measures invariance to the pretraining rewrite family under novel compositions rather than robustness to arbitrary semantics-preserving edits (Section 5). For clone detection, all InvPT models outperform their baselines on transformed data (Table 4), with relative gains up to 22.2%. InvGraphCodeBERT also surpasses both ContraBERT_C and ContraBERT_G on all four benchmarks; InvCodeBERT does so on Java250, Python800, and C++1400, but on POJ104 (62.03) it remains below ContraBERT_C (62.76) and ContraBERT_G (64.20).

For code classification, InvPT again yields consistent gains on transformed data (+T columns of Table 5), improving over GraphCodeBERT by up to 7.7% and over ContraBERT_G by up to 6.1%. Notably, these gains extend to C++1400 (up to +3.77 on classification and +5.84 on clone detection), again demonstrating *cross-language transfer* of structural robustness from Java and Python to a held-out target language. We do not claim cross-lingual representational alignment (*i.e.*, mapping semantically equivalent programs across languages into nearby embeddings), which would require parallel multi-language data and is left to future work.

Where InvPT helps least. The smallest robustness gain on the transformed test sets is InvContraBERT_C vs. ContraBERT_C on POJ104 (+0.77 pp, +1.2%), indicating that ContraBERT_C's prior contrastive objective already captures much of the invariance signal on this comparatively saturated benchmark. The gain is largest where the baseline is weakest: on transformed C++1400, where C/C++ never appears in our pretraining corpus, InvCodeBERT improves over CodeBERT by +5.84 pp (+22.8%).

Why cross-language transfer works. We attribute the transfer to two factors. First, our operators target language-agnostic structural concepts: loop equivalence (`W2F`, `F2W`), branch reversal (`RevIf`), scalar increment forms

Table 4. Clone detection results. "orig": original test set; "+T": transformed test set for robustness evaluation.

	Java250		Python800		C++1400		POJ104	
Model	orig	+ T	orig	+ T	orig	+ T	orig	+ T
CodeBERT	77.28	51.16	83.71	62.17	56.21	25.59	85.47	55.96
GraphCodeBERT	82.15	53.52	86.06	65.89	58.76	30.26	84.09	58.63
ContraBERT_C	82.19	49.64	85.60	63.41	57.65	27.09	89.58	62.76
ContraBERT_G	83.47	53.79	85.97	65.98	59.15	29.11	90.80	64.20
CodeSage	72.43	25.92	61.11	25.89	51.53	21.33	78.20	52.87
ModernBERT	80.51	41.06	83.16	52.85	59.05	24.66	88.22	52.86
InvCodeBERT	82.44	59.27 (+15.9%)	85.63	70.55 (+13.5%)	57.80	31.43 (+22.8%)	88.81	62.03 (+10.8%)
InvGraphCodeBERT	84.41	**62.30** (+16.4%)	**86.73**	73.91 (+12.2%)	**59.23**	**33.68** (+11.3%)	90.66	67.32 (+14.8%)
InvContraBERT_C	82.63	60.68 (+22.2%)	86.16	74.34 (+17.2%)	58.09	32.07 (+18.4%)	89.28	63.53 (+1.2%)
InvContraBERT_G	**84.53**	62.24 (+15.7%)	86.34	**74.60** (+13.1%)	59.23	33.10 (+13.7%)	**91.25**	**68.63** (+6.9%)

Table 5. Code classification results. "orig": original test set; "+T": transformed test set for robustness evaluation.

	Java250		Python800		C++1400		POJ104	
Model	orig	+ T	orig	+ T	orig	+ T	orig	+ T
CodeBERT	97.80	65.81	98.92	63.26	89.63	39.92	98.51	50.97
GraphCodeBERT	97.97	67.51	99.06	78.52	89.70	41.52	98.68	52.03
ContraBERT_C	97.90	67.33	98.89	72.61	89.54	41.46	98.48	51.19
ContraBERT_G	98.15	68.57	99.10	78.59	89.53	41.23	98.63	50.38
InvCodeBERT	98.16	70.10 (+6.5%)	99.00	82.42 (+30.3%)	89.94	43.69 (+9.4%)	98.63	53.67 (+5.3%)
InvGraphCodeBERT	98.08	71.30 (+5.6%)	99.09	**84.60** (+7.7%)	**90.00**	**44.23** (+6.5%)	98.54	52.66 (+1.2%)
InvContraBERT_C	98.07	71.35 (+6.0%)	98.95	83.28 (+14.7%)	89.87	42.51 (+2.5%)	98.72	53.24 (+4.0%)
InvContraBERT_G	**98.21**	**71.48** (+4.2%)	**99.11**	83.42 (+6.1%)	89.71	42.88 (+4.0%)	**98.73**	**53.74** (+6.7%)

(PP2AA, AA2EA), and identifier renaming (VarRe). Second, Java, Python, and C/C++ share a lexical substrate (common keywords, operators, and bracketing) that the encoder's sub-word tokenizer represents consistently across languages. We do not provide a theoretical analysis of this transfer; these two factors are the strongest explanation our evidence supports.

Absolute percentage-point improvements. The tables above report performance and relative improvement (%) over each baseline. For readers who prefer absolute changes, Table 6 and Table 7 restate the robustness gains on the transformed (+T) test sets in percentage points (pp), the simple arithmetic difference between each InvPT model and its corresponding baseline (*e.g.*, InvCodeBERT on Java250 (+T): a +8.11 pp gain over CodeBERT). InvPT improves in all 16 model–dataset comparisons on each task, but the distributions differ sharply. On clone detection the gains are uniformly sizable (median +8.07 pp, up to +11.04). On classification they are smaller and strongly skewed (median +3.56 pp), with the largest value (+19.16, InvCodeBERT on Python800) well separated from the next (+10.67); we therefore treat clone detection as the more representative measure and the classification maximum as an outlier rather than a typical gain. Measured against the degradation itself, InvPT recovers only part of the drop from the original to the transformed test set: a median of 29.6% on clone detection (range 2.9–49.3%) and 8.7% on classification (range 1.4–53.7%), so the gap that remains is larger than the part we close.

Table 6. Clone detection (+T): absolute pp change vs. the corresponding baseline encoder (InvCodeBERT vs. CodeBERT, InvGraphCodeBERT vs. GraphCodeBERT, etc.).

Model	Java250	Python800	C++1400	POJ104
InvCodeBERT	+8.11	+8.38	+5.84	+6.07
InvGraphCodeBERT	+8.78	+8.02	+3.42	+8.69
InvContraBERT_C	+11.04	+10.93	+4.98	+0.77
InvContraBERT_G	+8.45	+8.62	+3.99	+4.43

Table 7. Code classification (+T): absolute pp change vs. the corresponding baseline encoder.

Model	Java250	Python800	C++1400	POJ104
InvCodeBERT	+4.29	+19.16	+3.77	+2.70
InvGraphCodeBERT	+3.79	+6.08	+2.71	+0.63
InvContraBERT_C	+4.02	+10.67	+1.05	+2.05
InvContraBERT_G	+2.91	+4.83	+1.65	+3.36

3.4 Ablation Study

We ablate the contribution of each component of InvPT (Section 2) with three CodeBERT-based variants: 1. **w/o contrastive loss**: drops the supervised contrastive objective and trains with only MLM on original and transformed code; 2. **w/o self-contrast**: keeps only invariant-contrast pairs; 3. **InvCodeBERT + NL**: adds natural-language descriptions alongside code in pretraining. We evaluate the variants on clone detection (CodeNet, POJ104) with original and transformed (+T) test sets, and report the full result in Table 8.

Contrastive learning is the primary driver of improvement. Removing the contrastive loss and training only with MLM on original and transformed code still improves over CodeBERT on most benchmarks, but it remains clearly below full InvCodeBERT. On the original POJ104, it even drops below the CodeBERT baseline (83.87 vs. 85.47), indicating that exposure to transformed code without explicit representation alignment is insufficient.

Self-contrast provides consistent additional gains. Removing self-contrast lowers performance on all original benchmarks relative to InvCodeBERT, by roughly 0.5 to 1.1 points, and usually also reduces robustness on transformed test sets. The margins are modest, but the pattern is stable across datasets, suggesting that aligning different masked views of the same program complements invariant contrast rather than duplicating it.

Natural language descriptions are not necessary. Adding natural language descriptions yields mixed results. The effect is inconsistent across benchmarks and small relative to the other components: on the transformed sets, +NL gains +2.08 pp on both Java250 and POJ104 but loses 0.68 and 0.42 pp on Python800 and C++1400, averaging 0.76 pp above InvCodeBERT on transformed data (56.58 vs. 55.82) and 0.18 pp below it on original data (78.49 vs. 78.67). We therefore do not claim that code-only pretraining outperforms +NL. The ablation supports only a weaker claim: paired natural language moves performance by well under a point on average, and in opposite directions on the two settings. That gain does not justify a bi-modal NL-PL corpus, which restricts continued pretraining to the comparatively small set of functions carrying usable docstrings. Table 9 restates the ablation effects on the transformed (+T) test sets as absolute pp changes relative to full InvCodeBERT, making each removal's magnitude directly comparable: dropping the contrastive loss costs the most (−2.28 to −4.52 pp), while removing self-contrast or adding NL descriptions matters less.

3.5 Qualitative Representation Visualization

We next present a qualitative view of the learned representation space. Following Liu et al. [13], we randomly select five problems from the test split of POJ104, apply invariant transformations to the code snippets within these problems, and use all 500 code snippets per problem for visualization. We then apply t-SNE [39] to project the high-dimensional code representations into two dimensions. Figure 2 shows the results for two model families.

Across both families, the base models (CodeBERT and GraphCodeBERT) produce heavily overlapping embeddings with little cluster separation: invariant-transformed programs often lie closer to unrelated solutions than to their semantic equivalents. The ContraBERT variants (ContraBERT_C and ContraBERT_G) introduce more structure, with same-problem points forming local groupings, though considerable overlap remains. In contrast, our InvPT models (InvCodeBERT and InvGraphCodeBERT) yield tight, well-separated clusters for each problem, providing qualitative evidence consistent with the quantitative results.

4 Related Work

Pretrained code models. Code representation learning adapts encoder pretraining to programming languages, with models such as CodeBERT [5] and GraphCodeBERT [6] learning from large code corpora [3, 40]. Subsequent work adds AST sequences [41], data-flow graphs [6], execution signals [20], and transformation-based augmentation [11]. None of them explicitly trains for invariant embeddings under semantics-preserving transformations. Recent work confirms that encoders remain competitive and efficient for understanding tasks [7, 42, 34, 8], motivating InvPT's encoder-based design. A complementary line of work scales code embedding models well beyond the 125M-parameter class [43, 44, 45, 46, 47]; InvPT is architecture-agnostic; we instantiate it on the 125M-parameter encoders that dominate cost-sensitive deployments, and we include CodeSage [33] (1.3B) and ModernBERT [34] as larger reference points in Table 4.

Contrastive learning for code. Contrastive learning is a dominant paradigm for transferable representations; SimCLR [48] established that augmentation composition and a learnable projection head are critical ingredients. In the code domain, VarCLR [49] and CodeSage [33] show that contrasting transformed code improves representation quality, while NatGen [50] applies similar semantics-preserving transformations in a generative denoising setting.

Table 8. Ablation studies on clone detection using CodeNet and POJ104. "orig": original test set; "+T": transformed test set.

	Java250		Python800		C++1400		POJ104	
Model	orig	+ T	orig	+ T	orig	+ T	orig	+ T
CodeBERT	77.28	51.16	83.71	62.17	56.21	25.59	85.47	55.96
w/o contrastive loss	80.16	56.12	84.55	68.27	56.05	28.47	83.87	57.51
w/o self-contrast	81.92	57.92	85.07	70.27	57.19	30.33	87.69	62.42
InvCodeBERT + NL	82.74	61.35	85.52	69.87	57.69	31.01	88.02	64.11
InvCodeBERT	82.44	59.27	85.63	70.55	57.80	31.43	88.81	62.03

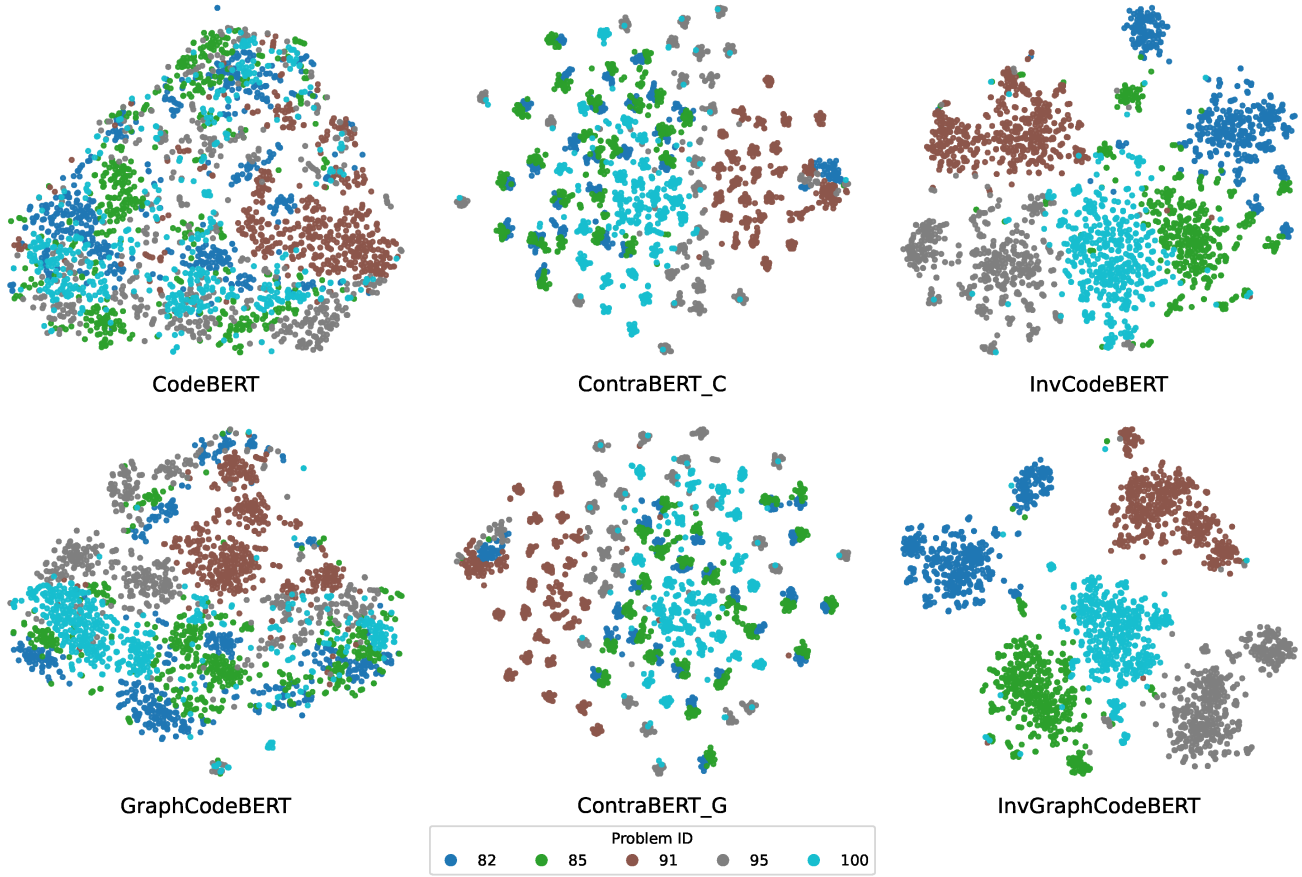

Figure 2. Visualization of vector embeddings of 5 problems in POJ104. Top row: CodeBERT-based models; bottom row: GraphCodeBERT-based models.

Table 9. Ablations (+T) absolute pp change.

Variant	Java250	Python800	C++1400	POJ104
w/o contrastive loss	−3.15	−2.28	−2.96	−4.52
w/o self-contrast	−1.35	−0.28	−1.10	+0.39
InvCodeBERT + NL	+2.08	−0.68	−0.42	+2.08

Two lines of work share our core premise that semantics-preserving transformations supply a contrastive signal for code: Corder [16] contrasts AST-level source-to-source variants, and ContraCode [51] contrasts compiler-generated JavaScript variants. We do not claim that premise as a contribution. InvPT differs in three respects. Both prior methods use pairwise InfoNCE, which treats two variants of the *same* function as negatives when they co-occur in a batch; since we apply several operators per function, such collisions are frequent, and we instead use a multi-positive supervised contrastive mask keyed on a hash of the canonical source

(Section 2.2). We apply the recipe as *continued* pretraining on released encoders rather than pretraining from scratch, and we evaluate robustness explicitly, on transformed test sets and a held-out target language, rather than standard performance alone. We do not rerun their objectives on our backbone, so we report no controlled comparison; our ablation isolates the contrastive objective as a whole (Section 3.4) but not the multi-positive mask against pairwise InfoNCE.

The closest prior work is ContraBERT [13], which combines MoCo with bi-modal NL-PL pretraining but evaluates robustness primarily on variable renaming. InvPT differs by using code-only continued pretraining with supervised contrastive learning over a shared encoder and a multi-positive mask, targeting naturally occurring semantics-preserving transformations (loop equivalence, branch reversal, scalar-increment forms, variable renaming) rather than dead-code insertion and random line deletion, and evaluating across multiple transformation types and downstream tasks. SPAT [11, 12] uses similar semantics-preserving transformations for fine-tuning augmentation; InvPT instead bakes them into pretraining as an objective.

Adversarial robustness. DAMP [52], MHM [53], and ALERT [9] use variable renaming and dead-code insertion as adversarial attacks on code models; Bielik and Vechev [54] give a systematic treatment of code adversarial robustness, Henkel et al. [55] formalize k-transformation robustness, and Rabin et al. [56] frame it as generalizability under natural transformations. Unlike defenses applied at fine-tuning time, InvPT builds invariance into pretraining, amortizing the benefit across downstream tasks.

5 Limitations

Our robustness evaluation uses test-time compositions that are unseen but built from the same six operators used in pretraining. The results therefore establish invariance to this rewrite family, not robustness to semantics-preserving change in general; independently generated transformations and naturally occurring refactorings remain untested. We also do not compare against downstream-only augmentation, so we cannot separate building invariance into pretraining from augmenting the fine-tuning data with the same rewrites.

All results come from a single pretraining and fine-tuning run per configuration, so differences of a point or less—including several ablation margins in Table 8—should be read with caution.

Our operators cover Java, Python, and C/C++; extending them to languages with substantially different semantics (*e.g.*, Haskell, Rust) requires non-trivial engineering to preserve equivalence. Because the rewrites are automated rather than formally verified, rare edge cases may alter runtime behavior, the clearest being `AA2EA` on Python (Section 2.1); we expect the affected fraction of our corpora to be small but have not measured it.

Finally, we evaluate discriminative tasks on competitive-programming benchmarks, which provide ground-truth labels but under-represent industrial codebases, and we instantiate InvPT on 125M-parameter encoder-only models pretrained on CodeSearchNet. Invariance is the appropriate criterion only for tasks whose output should not change under rewriting; generation requires the more general equivariance condition, which we leave to future work.

6 Conclusion

We present InvPT, a code-only continued pretraining method that combines semantics-preserving transformations with supervised contrastive learning and mixed-difficulty positive pairs, eliminating the need for paired natural-language data. InvPT improves robustness over its baseline in every model–dataset comparison on clone detection and code classification, by a median of 8.07 and 3.56 percentage points respectively (up to 11.04 and 19.16), while matching or improving standard task performance. These gains recover part of the degradation, not most of it, and are measured on compositions of the operator family seen in pretraining; robustness to structurally different rewrites remains open. Ablations show that invariant contrastive learning is the primary driver of these gains, with self-contrast providing consistent additional improvement. Notably, pretraining on Java and Python invariants alone yields robustness gains on C/C++ downstream tasks, suggesting that the learned invariance transfers across languages. More broadly, InvPT demonstrates that this form of robustness can be learned from code alone. Because the method modifies only the pretraining data and loss, extending it to richer transformation families and to encoder-decoder or decoder-only architectures is a natural next step. Generation, however, calls for a weaker criterion than invariance: under `VarRe` a model should rename consistently in its output rather than leave it unchanged. The general condition for a transformation T and a model g is equivariance, $g(T(x)) = M_T(g(x))$ for a transformation-specific output map M_T, with invariance the special case $M_T = \mathrm{id}$; learning equivariance under a known M_T is the direction we see as most promising.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This work is supported by the UC Noyce Initiative.

References

[1] Mark Chen et al. 2021. Evaluating large language models trained on code. (2021). https://arxiv.org/abs/2107.03374 arXiv: 2107.03374 [cs.LG].

[2] Baptiste Rozière et al. 2024. Code llama: open foundation models for code. (2024). https://arxiv.org/abs/2308.12950 arXiv: 2308.12950 [cs.CL].

[3] Yifeng He et al. 2024. Unitsyn: a large-scale dataset capable of enhancing the prowess of large language models for program testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2024). Association for Computing Machinery, Vienna, Austria, 1061–1072. ISBN: 9798400706127. doi:10.1145/3650212.3680342.

[4] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt fuzzing for fuzz driver generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. Salt Lake City, UT, USA, (Oct. 14–18, 2024). doi:10.1145/3658644.3670396.

[5] Zhangyin Feng et al. 2020. CodeBERT: a pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. doi:10.18653/v1/2020.findings-emnlp.139.

[6] Daya Guo et al. 2021. GraphCodeBERT: pre-training code representations with data flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ.

[7] Thennal D K, Tim Fischer, and Chris Biemann. 2025. Large language models are overparameterized text encoders. In *Proceedings of the 10th Workshop on Representation Learning for NLP (RepL4NLP-2025)*. Association for Computational Linguistics, Albuquerque, NM, (May 2025), 170–184. ISBN: 979-8-89176-245-9. doi:10.18653/v1/2025.repl4nlp-1.13.

[8] Yibin Lei, Shwai He, Ang Li, and Andrew Yates. 2026. Making large language models efficient dense retrievers. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, San Diego, California, United States, (July 2026), 12898–12913. ISBN: 979-8-89176-390-6. doi:10.18653/v1/2026.acl-long.587.

[9] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering* (ICSE '22). Association for Computing Machinery, Pittsburgh, Pennsylvania, 1482–1493. ISBN: 9781450392211. doi:10.1145/3510003.3510146.

[10] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering* (ICSE '22). ACM, (May 2022). doi:10.1145/3510003.3510049.

[11] Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data augmentation by program transformation. *Journal of Systems and Software*, 190, 111304. doi:10.1016/j.jss.2022.111304.

[12] Deze Wang et al. 2022. Bridging pre-trained models and downstream tasks for source code understanding. In *Proceedings of the 44th International Conference on Software Engineering* (ICSE '22). Association for Computing Machinery, Pittsburgh, Pennsylvania, 287–298. ISBN: 9781450392211. doi:10.1145/3510003.3510062.

[13] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. ContraBERT: enhancing code pre-trained models via contrastive learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2476–2487. doi:10.1109/ICSE48619.2023.00207.

[14] Naoto Ogura, Shinsuke Matsumoto, Hideaki Hata, and Shinji Kusumoto. 2018. Bring your own coding style. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 527–531. doi:10.1109/SANER.2018.8330253.

[15] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. 2019. The secret sharer: evaluating and testing unintended memorization in neural networks. In *28th USENIX Security Symposium (USENIX Security 19)* (SEC'19). USENIX Association, Santa Clara, CA, USA, 267–284. ISBN: 9781939133069. https://www.usenix.org/conference/usenixsecurity19/presentation/carlini.

[16] 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). Association for Computing Machinery, Virtual Event, Canada, 511–521. ISBN: 9781450380379. doi:10.1145/3404835.3462840.

[17] Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, and Hao Chen. 2023. Understanding programs by exploiting (fuzzing) test cases. In *Findings of the Association for Computational Linguistics: ACL 2023*. Association for Computational Linguistics, Toronto, Canada, (July 2023), 10667–10679. doi:10.18653/v1/2023.findings-acl.678.

[18] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 9726–9735. doi:10.1109/CVPR42600.2020.00975.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, (June 2019), 4171–4186. doi:10.18653/v1/N19-1423.

[20] Jiabo Huang et al. 2024. Code representation pre-training with complements from program executions. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*. Association for Computational Linguistics, Miami, Florida, US, (Nov. 2024), 267–278. doi:10.18653/v1/2024.emnlp-industry.21.

[21] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2019. Representation learning with contrastive predictive coding. (2019). https://arxiv.org/abs/1807.03748 arXiv: 1807.03748 [cs.LG].

[22] Prannay Khosla et al. 2020. Supervised contrastive learning. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (NIPS '20). Curran Associates Inc., Vancouver, BC, Canada. ISBN: 9781713829546.

[23] 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*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, (Nov. 2021), 6894–6910. doi:10.18653/v1/2021.emnlp-main.552.

[24] James Kirkpatrick et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114, 13, 3521–3526. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.1611835114. doi:10.1073/pnas.1611835114.

[25] Dayal Singh Kalra and Maissam Barkeshli. 2024. Why warmup the learning rate? underlying mechanisms and improvements. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=NVl4SAmz5c.

[26] 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 Thirtieth AAAI Conference on Artificial Intelligence* (AAAI'16). AAAI Press, Phoenix, Arizona, 1287–1293. doi:10.1609/aaai.v30i1.10139.

[27] Ruchir Puri et al. 2021. Codenet: a large-scale AI for code dataset for learning a diversity of coding tasks. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. https://openreview.net/forum?id=6vZVBkCDrHT.

[28] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 476–480. doi:10.1109/ICSME.2014.77.

[29] Jens Krinke and Chaiyong Ragkhitwetsagul. 2022. Bigclonebench considered harmful for machine learning. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, 1–7. doi:10.1109/IWSC55060.2022.00008.

[30] Jens Krinke and Chaiyong Ragkhitwetsagul. 2025. How the misuse of a dataset harmed semantic clone detection. (2025). https://arxiv.org/abs/2505.04311 arXiv: 2505.04311 [cs.SE].

[31] 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. In *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html.

[32] Yangruibo Ding et al. 2025. Vulnerability detection with code language models: how far are we? In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)* (ICSE), 1729–1741. doi:10.1109/ICSE55347.2025.00038.

[33] Dejiao Zhang et al. 2024. Code representation learning at scale. In *International Conference on Learning Representations*. https://proceedings.iclr.cc/paper_files/paper/2024/file/cfbba5249393100ada0bfb37557d2fd9-Paper-Conference.pdf.

[34] Benjamin Warner et al. 2025. Smarter, better, faster, longer: a modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vienna, Austria, (July 2025), 2526–2547. ISBN: 979-8-89176-251-0. doi:10.18653/v1/2025.acl-long.127.

[35] Shuai Lu et al. 2021. CodeXGLUE: a machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*. https://openreview.net/forum?id=6lE4dQXaUcb.

[36] Kevin Musgrave, Serge Belongie, and Ser-Nam Lim. 2020. A metric learning reality check. In *European Conference on Computer Vision*. Springer International Publishing, Cham, 681–699. ISBN: 978-3-030-58595-2. doi:10.1007/978-3-030-58595-2_41.

[37] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: evaluating the state of semantic code search. (2020). https://arxiv.org/abs/1909.09436 arXiv: 1909.09436 [cs.LG].

[38] Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Bkg6RiCqY7.

[39] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9, 86, 2579–2605. http://jmlr.org/papers/v9/vandermaaten08a.html.

[40] Yifeng He, Jicheng Wang, Yuyang Rong, and Hao Chen. 2025. FuzzAug: data augmentation by coverage-guided fuzzing for neural test generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*. Association for Computational Linguistics, Suzhou, China, (Nov. 2025), 15642–15655. ISBN: 979-8-89176-335-7. doi:10.18653/v1/2025.findings-emnlp.847.

[41] Daya Guo et al. 2022. UniXcoder: unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, (May 2022), 7212–7225. doi:10.18653/v1/2022.acl-long.499.

[42] Jiayi Lin, Yanlin Wang, Yibiao Yang, Lei Zhang, and Yutao Xie. 2026. Towards better code understanding in decoder-only models with contrastive learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40, 38, (Mar. 2026), 32006–32014. doi:10.1609/aaai.v40i38.40471.

[43] Daria Kryvosheieva, Saba Sturua, Michael Günther, and Han Xiao. 2025. Efficient code embeddings from code generation models. Jina Code Embeddings model series. (2025). https://arxiv.org/abs/2508.21290 arXiv: 2508.21290 [cs.CL].

[44] Yanzhao Zhang et al. 2025. Qwen3 embedding: advancing text embedding and reranking through foundation models. (2025). https://arxiv.org/abs/2506.05176 arXiv: 2506.05176 [cs.CL].

[45] Henrique Schechter Vera et al. 2025. EmbeddingGemma: powerful and lightweight text representations. (2025). https://arxiv.org/abs/2509.20354 arXiv: 2509.20354 [cs.CL].

[46] Tarun Suresh et al. 2024. CoRNStack: high-quality contrastive data for better code retrieval and reranking. CodeRankEmbed model. (2024). https://arxiv.org/abs/2412.01007 arXiv: 2412.01007 [cs.SE].

[47] Yue Wang et al. 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*. Association for Computational Linguistics, Singapore, (Dec. 2023), 1069–1088. doi:10.18653/v1/2023.emnlp-main.68.

[48] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *Proceedings of the 37th International Conference on Machine Learning* (Proceedings of Machine Learning Research). Vol. 119. PMLR, (13–18 Jul 2020), 1597–1607. https://proceedings.mlr.press/v119/chen20j.html.

[49] Qibin Chen et al. 2022. VarCLR: variable semantic representation pre-training via contrastive learning. In *Proceedings of the 44th International Conference on Software Engineering* (ICSE '22). Association for Computing Machinery, Pittsburgh, Pennsylvania, 2327–2339. ISBN: 9781450392211. doi:10.1145/3510003.3510162.

[50] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2022). Association for Computing Machinery, Singapore, Singapore, 18–30. ISBN: 9781450394130. doi:10.1145/3540250.3549162.

[51] Paras Jain et al. 2021. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, (Nov. 2021), 5954–5971. doi:10.18653/v1/2021.emnlp-main.482.

[52] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proc. ACM Program. Lang.*, 4, OOPSLA, (Nov. 2020). doi:10.1145/3428230.

[53] Huangzhao Zhang et al. 2020. Generating adversarial examples for holding robustness of source code processing models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34, 01, (Apr. 2020), 1169–1176. doi:10.1609/aaai.v34i01.5469.

[54] Pavol Bielik and Martin Vechev. 2020. Adversarial robustness for code. In *Proceedings of the 37th International Conference on Machine Learning* (Proceedings of Machine Learning Research). Vol. 119. PMLR, (13–18 Jul 2020), 896–907. https://proceedings.mlr.press/v119/bielik20a.html.

[55] Jordan Henkel et al. 2022. Semantic robustness of models of source code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 526–537. doi:10.1109/SANER53432.2022.00070.

[56] Md Rafiqul Islam Rabin et al. 2021. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135, 106552. doi:10.1016/j.infsof.2021.106552.

Received 2026-06-21; accepted 2026-08-14