POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING

MASTER's Degree Thesis

AI-based Automatic Generated Comments of Source Code

Supervisors

Prof. Antonio VETRÒ

Ing. Giacomo FANTINO

Candidate

Alessandro MANERA

DECEMBER 2025

Abstract

Automatic code comment generation is the task of producing concise natural-language summaries that describe the functionality, behavior, and intent of a given piece of source code. Such summaries support program comprehension, maintenance, and overall software quality.

As modern software systems grow in scale and complexity, automatic comment generation has become increasingly relevant in software engineering, where fast-paced development cycles and large codebases often lead to incomplete, outdated, or missing documentation. Transformer-based language models such as *CodeBERT* have achieved strong results by leveraging large-scale pretraining on code–text pairs, yet standard supervised fine-tuning typically emphasizes token-level patterns and may fail to capture deeper semantic properties of program behavior. This limitation frequently results in comments that are fluent but semantically shallow, offering limited insight into the logic and intent of the underlying code.

This thesis investigates whether *contrastive representation learning* can enhance the semantic quality of generated comments by reshaping the encoder's internal embedding space prior to generative fine-tuning. Two complementary contrastive strategies are explored. The first, *code–code alignment*, learns functional similarity between semantically equivalent Python snippets, promoting stable and intention-aware representations. The second, *code–diff alignment*, models fine-grained changes across consecutive code revisions extracted from version-control histories, aiming to capture the semantics of software evolution.

Both encoders are subsequently integrated into encoder–decoder (ED) architectures and fine-tuned for comment generation on the Python subset of the *CodeXGLUE* benchmark. A multi-phase training regime with staged unfreezing and continuation rounds is adopted to ensure stable optimization under limited computational resources. The experimental evaluation—spanning ROUGE-L, SacreBLEU, METEOR, and BERTScore—reveals that the baseline CodeBERT model remains strongest overall, while the contrastively trained code–code encoder achieves competitive performance and produces semantically coherent summaries. The diff-aware encoder, although conceptually promising, shows reduced generative fluency due to noise and partial context in commit-level data.

Despite mixed quantitative outcomes, the study demonstrates that contrastive objectives effectively structure code representations and can enrich encoder semantics in ways that extend beyond lexical patterns. The results highlight both the potential and the current limitations of applying contrastive and change-aware pretraining to code generation tasks, suggesting that larger, cleaner corpora and hybrid training objectives may be necessary to fully leverage these signals. Overall, this work contributes a reproducible methodological framework, a detailed empirical analysis, and a set of insights that inform future research on semantically grounded code understanding and automatic software documentation.

Table of Contents

Chapter 1

Introduction

1.1 Context and Motivation

In modern software engineering, *source code summarization* plays a crucial role in bridging the gap between formal code representations and human comprehension. Developers are constantly required to understand, maintain, and modify extensive and often unfamiliar codebases, sometimes written by others years earlier. The ability to quickly grasp the intent and behavior of a function or module is essential for effective debugging, feature development, and collaboration within teams. Concise summaries and descriptive comments can significantly improve code readability and serve as a cognitive bridge between implementation details and the developer's conceptual model of the system.

However, as software systems grow in scale and complexity, producing and maintaining high-quality documentation has become increasingly difficult and time-consuming. Teams working under strict deadlines frequently prioritize functionality and delivery over documentation quality, leading to missing, incomplete, or outdated comments. This absence of consistent, reliable documentation can slow down software evolution, increase the risk of introducing bugs, and ultimately raise maintenance costs over a system's lifecycle.

Code summarization refers to the task of generating natural-language descriptions that convey the functionality, purpose, and logic of a piece of source code. High-quality summaries can enhance program comprehension, reduce maintenance costs, and mitigate human error during software evolution [1]. Despite its recognized importance, many real-world repositories still lack meaningful or up-to-date comments. Developers either skip writing them due to time constraints or rely on automatically generated docstrings that merely replicate identifiers and parameter names, providing little real insight into program behavior. Manual commenting remains an expensive and error-prone process [2], which explains the growing need for automatic solutions.

In recent years, the software engineering community has devoted increasing attention to *automatic code summarization*, an interdisciplinary area that intersects software maintenance, machine learning, and natural language processing (NLP). The goal is not merely to produce syntactically correct sentences but to generate coherent,

semantically faithful explanations that help developers understand what a piece of code does and why it was implemented that way [3]. To reach that level of semantic faithfulness, models must learn to capture the behavior of code as transformations of data—how inputs are read, processed, and returned. This broader notion of *data-level semantics* extends summarization beyond textual translation toward the actual dynamics of computation.

1.2 Problem Definition and Challenges

Despite remarkable progress, automatic code summarization remains a challenging research problem. Unlike natural language text, source code is highly structured and governed by strict syntactic and semantic rules. It expresses precise computational logic but omits much of the human reasoning behind it. A successful model must therefore go beyond surface patterns and capture the underlying semantics of code.

Several major challenges persist in this domain:

- **Semantic alignment:** Mapping code to natural language requires connecting low-level syntactic constructs (loops, function calls, variable assignments) to high-level intentions (*filter elements*, *normalize data*, *update configuration*). This alignment is complex and often ambiguous.
- **Data-driven semantics:** Most summarization systems describe *what* a function appears to do syntactically, but few account for *how* it manipulates data. Capturing the semantics of data-intensive operations—what variables change, how states evolve, or how results are derived—is essential for meaningful documentation.
- **Data scarcity and imbalance:** Large, high-quality code–comment datasets are scarce. While open-source repositories provide abundant code, the associated comments are often noisy, incomplete, or stylistically inconsistent, making supervised learning difficult.
- **Contextual dependence:** Many snippets depend on external information such as class hierarchies, imported modules, or project-specific naming conventions that are not explicitly contained in the snippet itself, leading to incomplete representations of program semantics.
- **Generalization:** Models trained on one language, domain, or repository may fail to generalize to others due to differences in syntax, style, or programming paradigms.
- **Evaluation difficulty:** Traditional metrics such as BLEU, METEOR, or ROUGE assess lexical similarity rather than semantic faithfulness, and therefore do not always reflect whether the generated summary truly conveys the intended meaning of the code.

These issues highlight a broader limitation of purely supervised text generation: models often learn superficial token correlations instead of true semantic understanding. Addressing these challenges requires learning mechanisms able to represent not only syntactic structure but also the effect of code execution on data. Representing *differences* between code fragments—the *code diff*—offers a way to approximate these behavioral effects. A diff can describe the addition, deletion, or modification of statements as well as the changes they induce on variables and control flow. Figure 1.1 illustrates how code diffs expose meaningful behavioral changes between revisions, making them valuable for learning representations that capture fine-grained semantic evolution. By learning from diffs, a model gains exposure to semantic variations: it sees how small textual edits can translate into major behavioral shifts or, conversely, how different implementations can yield equivalent outcomes. This perspective opens the door to modeling code semantics through its observable data transformations rather than its static form.

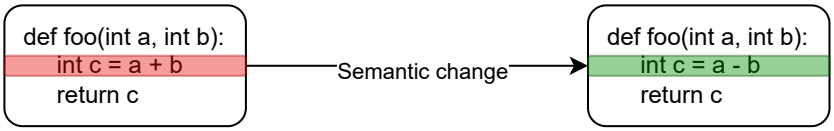


Figure 1.1: Example of a code diff highlighting behavioral changes between two function revisions. Learning from diffs helps the model capture fine-grained semantic evolution beyond superficial syntax.

1.3 The Role of Artificial Intelligence

The rapid advancement of deep learning and transformer-based architectures has profoundly reshaped the field of code intelligence. Large language models (LLMs) now demonstrate an unprecedented ability to process, understand, and generate both source code and natural language, narrowing the gap between programming and linguistic reasoning. This convergence has enabled models to capture complex code semantics and generate human-readable outputs with remarkable fluency.

The key innovation underlying this progress lies in the transformer architecture [4], which introduced self-attention as a mechanism for modeling long-range dependencies. Leveraging this design, large-scale pre-trained models such as *CodeBERT* [5], *CodeT5* [6], and *PLBART* [7] have become central components of modern code-understanding pipelines. Pre-trained on massive corpora of code–text pairs from repositories like GitHub, these models learn to encode both syntactic structure and semantic intent across multiple programming languages [8, 9]. They serve as general-purpose encoders that can be fine-tuned for downstream tasks such as code search, classification, or summarization.

However, while pretraining yields powerful general representations, downstream fine-tuning is crucial for task-specific specialization. Conventional supervised fine-tuning aligns code directly with its paired comment, but this approach often overfits to stylistic patterns in the dataset, producing grammatically fluent yet semantically

shallow summaries [10]. Moreover, such models tend to imitate lexical tokens without capturing the deeper logic of computation. To overcome this limitation, recent studies have explored *contrastive learning* [11, 12, 13], a self-supervised paradigm that structures the embedding space around semantic similarity. By pulling related examples closer and pushing unrelated ones apart, contrastive objectives teach the encoder to recognize functional equivalence across syntactically different programs. When applied to code, they promote embeddings that represent *intent* and *behavior*, not just textual co-occurrence. This strategy has already shown success in code retrieval, clone detection, and defect prediction, and it has recently begun to influence generative settings such as automatic documentation and comment generation. Figure 1.2 provides a conceptual illustration of the contrastive learning objective used during encoder pretraining, showing how positive and negative pairs shape the semantic structure of the embedding space.

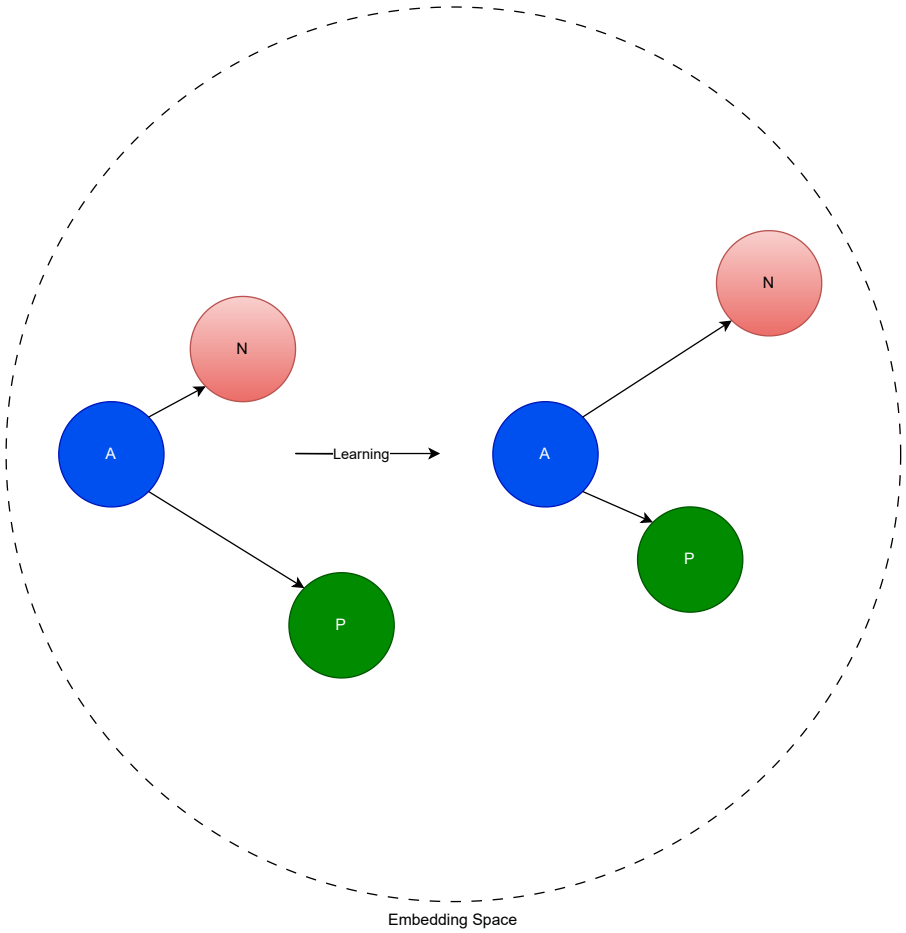


Figure 1.2: Illustration of the contrastive learning objective. Positive code pairs are pulled closer in the embedding space, while negative pairs are pushed apart, promoting semantic structuring of representations.

At the same time, modeling the *dynamics* of code evolution through code diffs introduces complementary information that contrastive learning alone cannot capture.

While contrastive objectives shape a global semantic geometry, diff-based training exposes the model to local transformations that directly reflect how data flow and program state change between versions. Combining the two allows the encoder to develop a richer representation of behavior: contrastive signals enforce global semantic alignment, whereas diff signals encode fine-grained data-centric semantics. Together, they move the task from mere textual translation toward an integrated understanding of code as a system that transforms data.

1.4 Objectives and Contributions

The present work investigates how these two paradigms—contrastive learning and diff-based semantic modeling—can be combined to improve automatic code comment generation. The study builds upon `CodeBERT` as a pre-trained foundation and introduces a fine-tuning pipeline in which the encoder is first enhanced through semantically structured contrastive objectives and then adapted to comment generation within an encoder–decoder (ED) framework. The goal is to produce summaries that are both linguistically coherent and semantically faithful to the data transformations performed by the code.

Three encoder variants are explored:

1. A **Base Encoder**, fine-tuned directly on standard code–comment pairs using a supervised objective;

2. A **Code–Code Encoder**, trained on semantically equivalent code snippets as positive samples to reinforce structural and functional awareness through contrastive learning;

3. A **Code–Diff Encoder**, trained on pairs of code edits extracted from version-control systems to model the semantics of incremental code changes and their effects on data.

The encoder variants are subsequently integrated into encoder–decoder architectures that generate comments conditioned on the learned representations. Comparative evaluation assesses how the inclusion of contrastive and diff-based training influences both lexical and semantic quality metrics. Quantitative results (ROUGE-L, SacreBLEU, METEOR, BERTScore) are complemented by qualitative analyses highlighting differences in comment completeness and behavioral accuracy.

Beyond empirical evaluation, the study contributes a broader perspective on how learning paradigms developed for natural-language understanding can be adapted to software artifacts. By uniting contrastive learning with data-centric representations derived from code diffs, the work aims to shift automatic summarization toward models that describe what code *does*—its impact on data and state—rather than merely what it *looks like*. This orientation toward behavior-aware documentation represents a step forward in the pursuit of semantically grounded code intelligence. To provide a clearer visual summary of the proposed approach, the overall training

and fine-tuning pipeline is illustrated in Figure 1.3. The diagram highlights the two-stage contrastive pretraining strategy and its integration into the encoder–decoder architecture for comment generation.

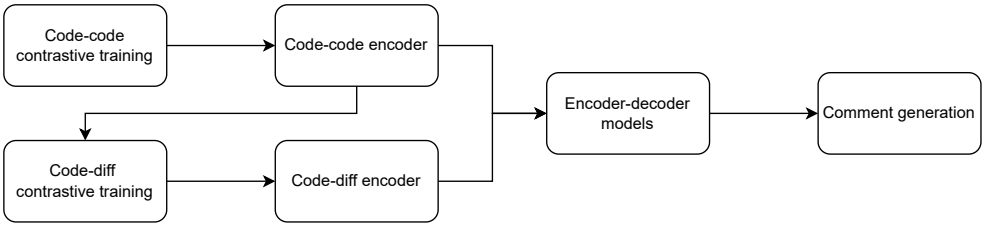


Figure 1.3: High-level overview of the proposed pipeline. Contrastive pretraining produces two specialized encoders (code–code and code–diff), which are subsequently integrated into encoder–decoder architectures for downstream comment generation.

Chapter 2

Related Works

Research in automatic code summarization builds upon decades of progress in the broader field of source code understanding and representation learning. The intersection of software engineering and natural language processing has gradually evolved from syntactic analysis toward deep semantic modeling, enabling machines to reason about the meaning and intent of code. The study of this evolution highlights how advances in representation, pretraining objectives, and contrastive methods have shaped current approaches to code–text alignment and comment generation.

2.1 Code Representation Learning

Machine learning models designed to process source code must transform it into numerical vectors while preserving its syntactic structure and semantic intent. Early approaches relied on *hand-crafted features* extracted from Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), or Program Dependence Graphs (PDGs) to encode the structural hierarchy of code elements [14, 15]. Although these symbolic representations captured some syntactic regularities, they required manual feature engineering and lacked the ability to generalize across programming languages and styles.

The emergence of deep learning shifted this paradigm by enabling data-driven feature extraction. Early neural approaches modeled source code as token sequences and employed recurrent neural networks (RNNs) and long short-term memory (LSTM) architectures to capture local dependencies [16]. These methods improved code summarization and classification by learning distributed code embeddings. However, because code differs from natural language in its hierarchical and compositional structure, sequential encoders often struggled to capture non-local relationships such as variable scoping, function dependencies, or nested control logic.

To address these limitations, researchers proposed structure-aware representations that explicitly leverage the hierarchical nature of code. For instance, tree-based and graph-based encoders, such as TreeLSTMs and Graph Neural Networks (GNNs), were used to propagate information across AST nodes and program graphs [17, 18]. These models improved semantic understanding by preserving relationships between

tokens beyond linear order. Nevertheless, they remained limited by scalability and training complexity, especially when dealing with real-world repositories containing millions of lines of code.

2.2 Transformer Models for Code Understanding

The introduction of the transformer architecture [4] marked a turning point in representation learning for both text and code. Its self-attention mechanism allowed models to efficiently capture long-range dependencies without relying on recurrence, leading to remarkable improvements in translation, summarization, and language understanding. These advances naturally extended to source code modeling, where the transformer's ability to handle variable-length sequences and contextualize tokens across arbitrary spans proved especially advantageous.

Pioneering models such as *CodeBERT* [5], *GraphCodeBERT* [13], and *CodeT5* [6] adapted large-scale language pretraining strategies to the domain of programming languages. These models were trained on massive corpora of code–text pairs from datasets like *CodeSearchNet* [19], jointly learning to represent both code and natural language descriptions through masked language modeling and text–code alignment objectives. By pretraining on multimodal data (code and comments), such models internalized cross-domain relationships that could later be transferred to downstream tasks.

A simplified representation of the transformer architecture used in most modern code models is shown in Figure 2.1.

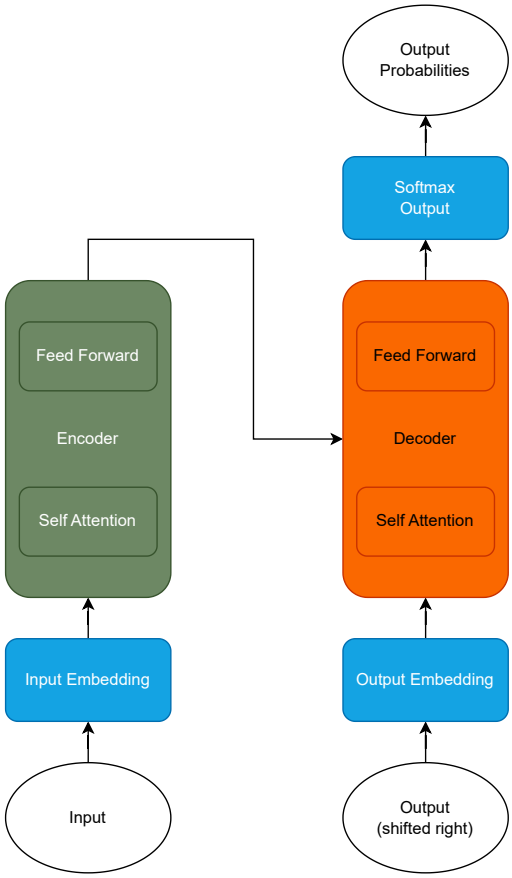


Figure 2.1: Simplified view of the transformer architecture, based on stacked self-attention and feed-forward layers. This design forms the backbone of most recent code understanding and generation models.

Transformer-based encoders provided a unified backbone for numerous applications, including code search, clone detection, defect prediction, and code summarization. Later extensions such as *PLBART* [7], *CodeT5+* [20], and *UniXCoder* [9] further generalized the pretraining paradigm by combining encoder–decoder objectives and cross-modal learning strategies. These models demonstrated improved robustness and generalization across programming languages and repositories, cementing transformers as the de facto architecture for code understanding and generation.

2.3 Contrastive and Retrieval-Based Learning

Although supervised fine-tuning on code–comment pairs has proven effective, it often suffers from data noise and limited semantic coverage. To overcome these limitations, researchers have increasingly turned to *contrastive learning*—a self-supervised paradigm that aligns representations through relative similarity rather than explicit labels. In this framework, the model learns to bring semantically related

samples closer together in the embedding space while pushing unrelated samples apart [11].

Contrastive learning has shown exceptional promise for software engineering tasks. For instance, *CodeContrast* [21] introduced contrastive objectives for cross-language code representation, while *CLSE* [12] demonstrated that contrastive encoders can outperform purely supervised baselines on retrieval and classification. These methods not only produce embeddings that are more discriminative but also mitigate overfitting by learning from implicit relational information present in large unlabeled datasets.

In the context of code–text alignment, contrastive learning enhances the encoder's ability to distinguish between correct and incorrect comment–code associations. This property makes it particularly relevant to automatic comment generation, where semantic precision matters as much as linguistic fluency. Related paradigms such as retrieval-augmented generation (RAG) [22] further extend this idea by combining pretrained encoders with retrieval mechanisms, enabling models to ground their output in real examples rather than relying solely on parametric memory.

2.4 Automatic Comment Generation

Automatic comment generation, also referred to as *code summarization*, aims to produce concise natural-language descriptions that capture the intent and functionality of a given code snippet. Early approaches framed this task as a sequence-to-sequence problem, training neural encoder–decoder models with attention mechanisms to translate code into human-readable text [16, 2]. These methods demonstrated that deep learning could automatically extract functional meaning from code, though their generalization remained limited by data availability and domain variability.

The transformer revolution significantly advanced comment generation by enabling large-scale pretraining and transfer learning. Models such as *PLBART* [7] and *CodeT5* [6] achieved state-of-the-art results by jointly optimizing for both code understanding and generation objectives. More recently, *CodeT5+* [20] extended this approach to multi-task settings, unifying summarization, translation, and refinement within a single framework. Surveys such as *Zhang et al. (2024)* [3] provide a detailed overview of the evolution of this field, including benchmark datasets, evaluation metrics, and architectural trends.

To contextualize the models surveyed in the literature, Figure 2.2 provides a concise depiction of the encoder–decoder paradigm commonly used for code summarization.

Figure 2.2: Typical encoder–decoder framework for code summarization. The encoder maps source code into a latent representation, while the decoder generates a natural-language summary conditioned on the encoded semantics.

Despite substantial progress, comment generation remains far from solved. Models frequently produce fluent but semantically inaccurate comments, failing to capture fine-grained program intent or logic-level nuances [23]. These issues are aggravated by the inherent ambiguity of natural language and the limited reliability of lexical metrics such as BLEU or ROUGE, which often overlook semantic adequacy. To address these gaps, recent studies have begun to integrate contrastive pretraining, retrieval-based reasoning, and semantic similarity scoring into the training pipeline [24, 22]. Such hybrid approaches aim to ensure that generated comments are not only grammatically correct but also faithfully aligned with the underlying code behavior.

Overall, the evolution of code representation and comment generation models reflects a broader shift from syntactic to semantic modeling, from supervised to self-supervised learning, and from isolated tasks to integrated multimodal frameworks. Yet, ensuring the *faithfulness*, *generalizability*, and *interpretability* of generated comments remains an open research challenge. Addressing these gaps provides the foundation and motivation for the present study, which investigates how contrastive fine-tuning can enhance semantic alignment between code and text representations.

Chapter 3

Methodology

3.1 Conceptual Framework and Objectives

The development of an automatic system for code comment generation requires bridging two fundamentally different representational domains: the formal, deterministic structure of source code and the flexible, context-sensitive structure of natural language. The methodological design adopted in this work is guided by the intuition that meaningful comment generation is only possible if the model acquires, in stages, a progressively refined internal understanding of program semantics. Before a system can *describe* code, it must first learn to *represent* it, and before representation it must learn to *distinguish* meaningful relationships among code fragments.

The proposed pipeline therefore follows a hierarchical paradigm composed of two macro-phases:

1. **Representation learning through contrastive pretraining**, designed to teach an encoder how to structure the embedding space around semantic relationships.
2. **Generative fine-tuning through encoder–decoder training**, where the pretrained encoder is integrated into a sequence-to-sequence model that learns to transform code into natural-language docstrings.

This sequence reflects a progression that mirrors a human developer's learning process. First, the encoder is trained to measure similarity between different fragments of code. Next, it observes how code transforms over time through version-control diffs, learning to recognize semantic changes. Finally, the encoder serves as the semantic backbone of a generative model that produces explanations conditioned on those learned representations.

Two complementary hypotheses motivate this methodology. First, that contrastive learning, by maximizing agreement between semantically similar code fragments, yields embeddings that better encode program-level intent than purely token-based pretraining. Second, that modeling the temporal evolution of code across commits further enriches this representation by exposing the encoder to explicit behavioral

variations. These ideas lead to the construction of two specialized encoders—one trained on function-level semantic equivalence (*code–code*) and one trained on commit-level revisions (*code–diff*)—later evaluated within a unified comment-generation framework.

3.2 Contrastive Representation Learning

Contrastive learning provides a self-supervised way to enforce a geometric notion of semantic similarity. Given an anchor example a, a positive example p conveying similar semantics, and a negative example n drawn from a different context, the encoder is trained to minimize the distance between a and p while maximizing the distance between a and n. The objective is expressed through the standard margin-based contrastive loss:

$$\mathcal{L}_{\text{ctr}} = \max\big(0,\, d(a,p) - d(a,n) + m\big), \tag{3.1}$$

where $d(\cdot)$ is the cosine distance and m is a fixed margin enforcing the required separation.

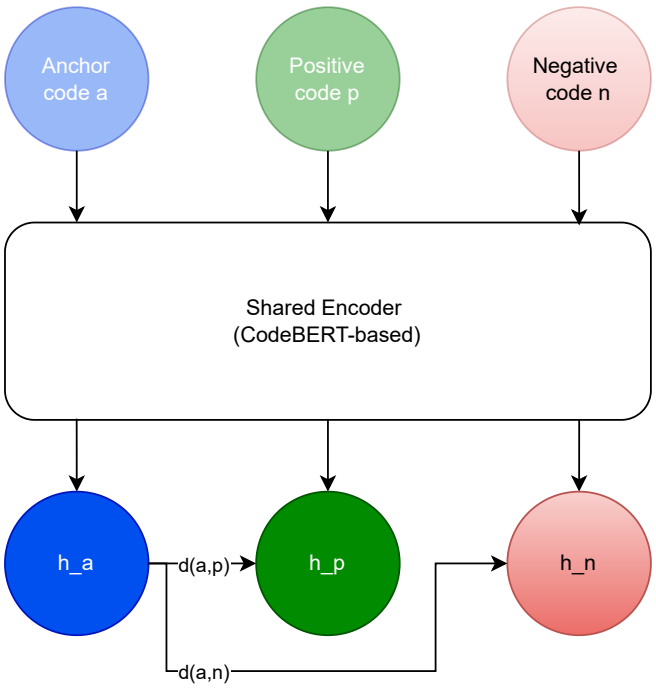


Figure 3.1: Contrastive triplet mechanism used during encoder pretraining. The encoder learns to map semantically related functions close together while pushing unrelated ones apart.

Figure 3.1 illustrates the core training dynamic: the anchor and positive examples are encouraged to form a compact cluster, while the negative example is repelled. Over

many iterations, this shapes an embedding space that reflects functional semantics rather than surface similarity.

3.2.1 Stage 1: Function-Level Code–Code Encoder

The first encoder is pretrained on the *Nan-Do/code-search-net-python* dataset, comprising Python function definitions paired with docstrings. Positive examples are selected through docstring similarity and lexical overlap, while negatives are drawn from unrelated samples in the same batch. Through this process, the encoder learns to recognize function-level equivalence even in the presence of syntactic variability.

This stage provides a foundation of structural and semantic understanding. Because many functions achieve similar goals through different control structures or naming conventions, contrastive learning encourages the encoder to abstract away from superficial patterns and focus instead on underlying behavior.

3.2.2 Stage 2: Commit-Level Code–Diff Encoder

The second encoder expands this notion of semantic understanding by modeling *how* code changes. It is initialized from the pretrained code–code model and further trained on the *python-state-changes* dataset, which contains pairs of functions $(v_{\text{old}}, v_{\text{new}})$ extracted from consecutive commits.

Unlike static function equivalence, commit-level training exposes the encoder to directional changes: bug fixes, refactorings, feature additions, and other forms of program evolution. The contrastive loss of Equation 3.1 is reused, but now the distance $d(v_{\text{old}}, v_{\text{new}})$ serves as a proxy for semantic modification across versions.

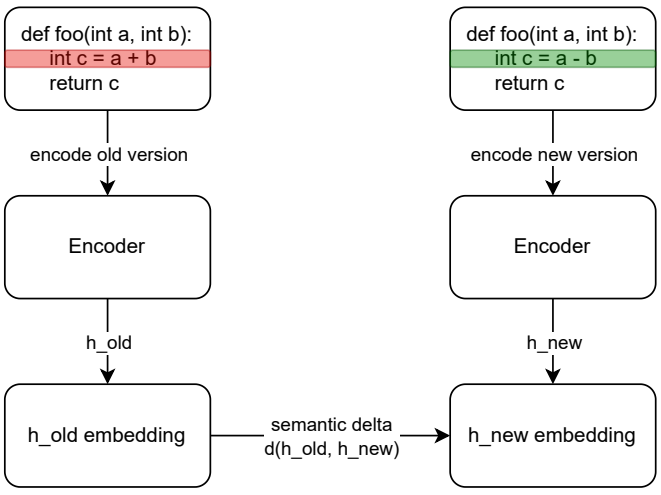


Figure 3.2: Commit-level semantic modeling. By comparing old and new function versions, the encoder learns representations that reflect the magnitude and nature of code changes.

Figure 3.2 illustrates this process: modifications in the abstract syntax, control

flow, or data handling result in corresponding displacement within the embedding space. The design mirrors transfer learning principles: the diff-aware encoder inherits structural understanding from Stage 1 and focuses its capacity on modeling *semantic evolution*.

3.3 Generative Fine-Tuning and Progressive Optimization

Once pretrained, each encoder is integrated into an encoder–decoder (ED) architecture for docstring generation. The decoder is initialized from `microsoft/codebert-base` in all configurations, ensuring comparable linguistic fluency across model variants.

Training minimizes the cross-entropy loss:

$$\mathcal{L}_{\text{gen}} = -\sum_{t=1}^{T} \log P_\theta(y_t \mid y_{<t}, x), \tag{3.2}$$

where x is the input code snippet and y_t the t-th target token. Equation 3.2 encourages the decoder to generate fluent natural-language descriptions grounded in the encoder's semantic representation.

3.3.1 Multistage Training Regime

Training follows a progressive schedule inspired by curriculum learning and stability optimization:

- **Phase 1: Fine-tuning** — moderate learning rates (2×10^{-5} to 3×10^{-5}) bring the model into the generative regime.
- **Phase 2: Continuation/Polishing** — lower learning rates (2×10^{-6} to 3×10^{-6}) refine the alignment between encoder semantics and language generation.
- **Selective Unfreezing** — Stage A freezes the encoder and trains only the decoder; Stage B unfreezes the top two encoder layers to allow controlled co-adaptation.

This schedule mitigates catastrophic forgetting and preserves the structure learned during contrastive pretraining.

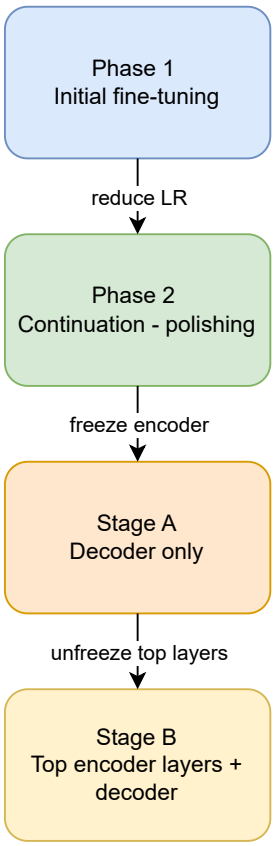


Figure 3.3: Progressive training schedule: initial full-model fine-tuning, low-learning-rate continuation phases, and selective encoder unfreezing (Stage A → Stage B).

3.3.2 Inference Strategy

Inference uses beam search with 4–6 beams, early stopping, a repetition penalty between 1.05 and 1.10, and a length penalty of 1.15 to balance conciseness and completeness. These settings reduce repetitive phrasing and encourage descriptive summaries consistent with the CodeXGLUE dataset.

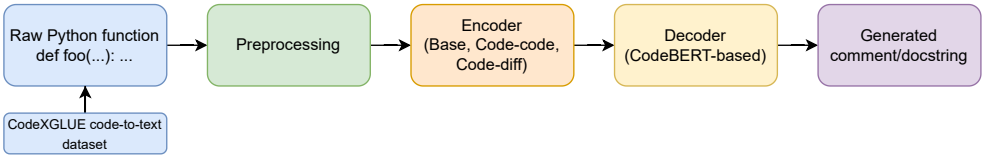


Figure 3.4: End-to-end encoder–decoder pipeline for code comment generation. The encoder provides a semantic representation that conditions the decoder's natural language output.

3.4 Implementation Details and Practical Limitations

All experiments were implemented in `PyTorch` using the Hugging Face `Transformers` and `Datasets` libraries, ensuring reproducibility and standardized model management. Training employed AdamW during the main fine-tuning phases and Adafactor during memory-constrained continuation stages, striking a balance between stability and GPU feasibility. To accommodate the limited hardware (a single 8 GB GPU), several memory-saving strategies were applied: mixed-precision computation (FP16), gradient checkpointing, and PyTorch's expandable CUDA segments to mitigate fragmentation.

Input code sequences were truncated at 256 tokens and corresponding target docstrings at 64 tokens, reflecting a compromise between representational coverage and GPU limits. Effective batch sizes ranged from 1 to 8, with gradient accumulation up to 16 steps to simulate larger batches. Depending on the model variant and available VRAM, each training phase spanned one to three epochs. In practice, training curves indicated continued improvement even at the final epoch, suggesting that full convergence was not reached under the available compute budget.

These constraints shape both the methodology and the interpretation of results. Sequential pretraining introduces dependencies across stages, meaning that weaknesses or noise in the code–code encoder propagate into the diff-aware encoder. Limited batch diversity may reduce the stability of contrastive optimization, and the reduced number of epochs restricts the decoder's ability to fully adapt to the embedding space provided by the pretrained encoder. Moreover, classical lexical metrics such as BLEU, ROUGE, and METEOR tend to underrepresent semantic adequacy, penalizing paraphrastic or stylistically diverse outputs that are nonetheless faithful to the intent of the code.

Despite these limitations, the overall pipeline remains coherent, reproducible, and structurally well-motivated. By integrating representational learning (contrastive stages), temporal semantics (commit-level deltas), and generative fine-tuning (encoder–decoder training), the methodology offers a unified framework for examining how different forms of pretraining influence downstream comment generation. The practical experience gained from training under realistic constraints further highlights the challenges and trade-offs inherent in medium-scale software-language modeling.

Chapter 4

Experiments

4.1 Overview

This chapter presents an extensive empirical assessment of the proposed models for automatic code comment generation. The experiments were designed to examine whether contrastive pretraining at the function level and change-aware pretraining at the commit level can improve downstream generation quality when compared with a baseline encoder–decoder initialized directly from `microsoft/codebert-base`. Three variants were therefore evaluated under identical fine-tuning and decoding conditions: **ED-Base**, **ED-Code-Code**, and **ED-Code-Diff**. The analysis integrates quantitative benchmarks with qualitative inspection of generated outputs, providing a comprehensive view of model behavior.

The central aim of this evaluation is twofold. First, to measure whether contrastively aligned embeddings contribute to more semantically faithful summaries of code. Second, to assess whether introducing temporal information from version control commits—capturing how code *changes*—produces richer or more contextually grounded documentation. The experiments also reveal the challenges of scaling such architectures under realistic computational constraints, highlighting the delicate balance between representational power, data quality, and training stability.

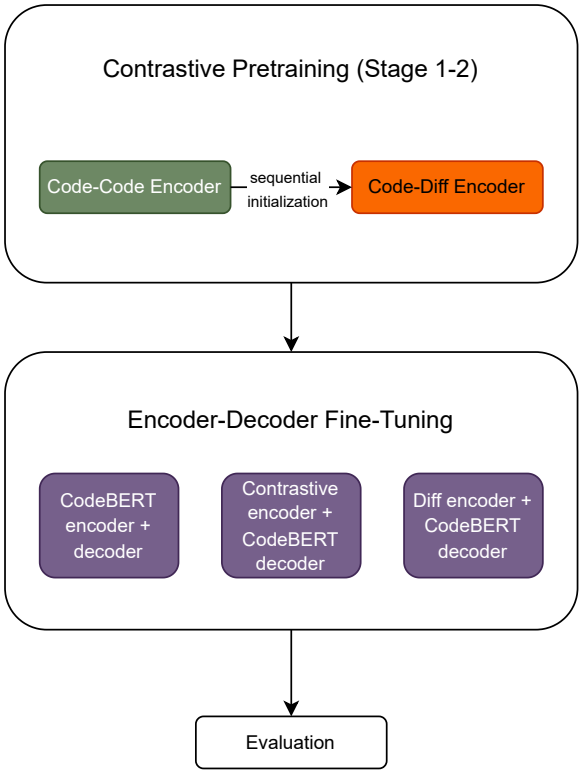


Figure 4.1: Summary of the three evaluated encoder–decoder models and their training dependencies.

4.2 Experimental Setup

Datasets

The learning pipeline developed in this work relies on three distinct datasets, each supporting a different stage of the methodology: (1) contrastive function-level pretraining, (2) contrastive diff-level pretraining, and (3) supervised encoder–decoder training for comment generation. Although all datasets contain Python code, they differ substantially in structure, supervision signals, and data quality. A clear understanding of these differences is essential for interpreting the empirical results.

CodeSearchNet Python: Function-Level Supervision

The *Nan-Do/code-search-net-python* corpus is a curated subset of the original CodeSearchNet dataset, containing Python function definitions paired with their corresponding docstrings. Each instance includes a complete, syntactically valid function body and a human-written summary, forming well-aligned supervision for learning semantic equivalence between code snippets.

Figure 4.2 shows a representative entry from this dataset.

code string · *lengths* 75 104k	code_tokens list	docstring string · *lengths* 1 46.9k
def ReplaceInFile(filename, old, new, encoding=None): ''' Replaces all occurrences of…	["def", "ReplaceInFile", "(", "filename", ",", "old", ",", "new", ","…	Replaces all occurrences of "old" by "new" in the given file. :param unicode filename: The name of…
def CreateDirectory(directory): ''' Create directory including any missing intermediate…	["def", "CreateDirectory", "(", "directory", ")", ":", "from", "six",…	Create directory including any missing intermediate directory. :param unicode directory:…
def DeleteDirectory(directory, skip_on_error=False): ''' Deletes a directory.…	["def", "DeleteDirectory", "(", "directory", ",", "skip_on_error", "=",…	Deletes a directory. :param unicode directory: :param bool skip_on_error: If True, ignore any…
def GetMTime(path): ''' :param unicode path: Path to file or directory :rtype: float :returns:…	["def", "GetMTime", "(", "path", ")", ":", "_AssertIsLocal", "(", "path", ")"…	:param unicode path: Path to file or directory :rtype: float :returns: Modification time for…
def ListMappedNetworkDrives(): ''' On Windows, returns a list of mapped network drives :return:…	["def", "ListMappedNetworkDrives", "(", ")", ":", "if", "sys", ".", "platform",…	On Windows, returns a list of mapped network drives :return: tuple(string, string, bool) For…
def CreateLink(target_path, link_path, override=True): ''' Create a symbolic link at…	["def", "CreateLink", "(", "target_path", ",", "link_path", ",",…	Create a symbolic link at `link_path` pointing to `target_path`. :param unicode target_path: Link…
def IsLink(path): ''' :param unicode path: Path being tested :returns bool: True if `path` is a…	["def", "IsLink", "(", "path", ")", ":", "_AssertIsLocal", "(", "path", ")", "if…	:param unicode path: Path being tested :returns bool: True if `path` is a link
def ReadLink(path): ''' Read the target of the symbolic link at `path`. :param unicode path: Pat…	["def", "ReadLink", "(", "path", ")", ":", "_AssertIsLocal", "(", "path", ")"…	Read the target of the symbolic link at `path`. :param unicode path: Path to a symbolic link…
def _AssertIsLocal(path): ''' Checks if a given path is local, raise an exception if not. This is…	["def", "_AssertIsLocal", "(", "path", ")", ":", "from", "six", ".", "moves",…	Checks if a given path is local, raise an exception if not. This is used in filesystem…

Figure 4.2: Example entry from the `CodeSearchNet` Python dataset. Each sample consists of a full function definition and its corresponding docstring, enabling clean function-level contrastive supervision.

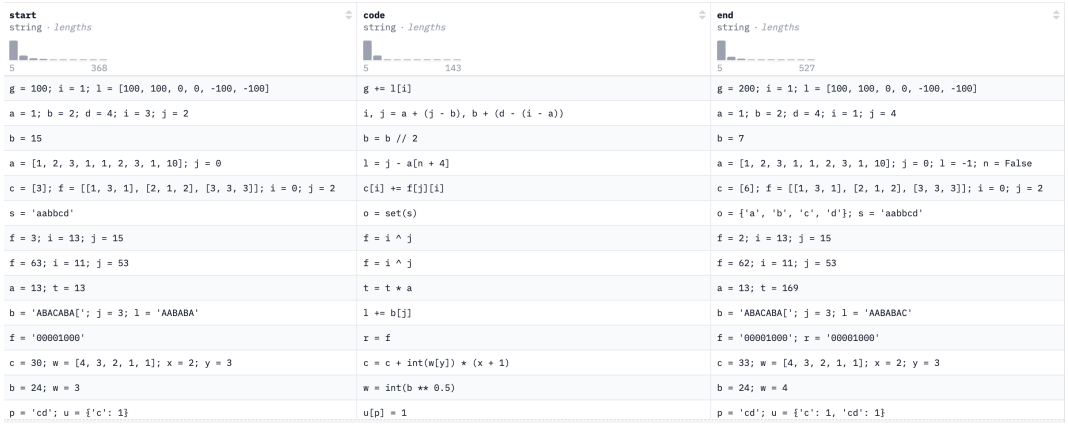
CodeSearchNet is particularly suitable for contrastive learning because different functions can implement similar behavior using diverse programming styles. Its primary limitation lies in stylistic variability: docstrings may differ in verbosity and quality, though they typically remain semantically meaningful.

Python State Changes: Diff-Level Supervision

The *python-state-changes* dataset provides fine-grained evolution traces of Python programs extracted from consecutive Git commits. Each entry contains three fields:

- `start`: the program state before the edit,
- `code`: the code fragment representing the edit applied in the commit,
- `end`: the program state after the edit.

Figure 4.3 illustrates this format, where updates range from arithmetic changes to structural modifications.



start string · lengths 5 368	code string · lengths 5 143	end string · lengths 5 527
g = 100; i = 1; l = [100, 100, 0, 0, -100, -100]	g += l[i]	g = 200; i = 1; l = [100, 100, 0, 0, -100, -100]
a = 1; b = 2; d = 4; i = 3; j = 2	i, j = a + (j - b), b + (d - (i - a))	a = 1; b = 2; d = 4; i = 1; j = 4
b = 15	b = b // 2	b = 7
a = [1, 2, 3, 1, 1, 2, 3, 1, 10]; j = 0	l = j - a[n + 4]	a = [1, 2, 3, 1, 1, 2, 3, 1, 10]; j = 0; l = -1; n = False
c = [3]; f = [[1, 3, 1], [2, 1, 2], [3, 3, 3]]; i = 0; j = 2	c[i] += f[j][i]	c = [6]; f = [[1, 3, 1], [2, 1, 2], [3, 3, 3]]; i = 0; j = 2
s = 'aabbcd'	o = set(s)	o = {'a', 'b', 'c', 'd'}; s = 'aabbcd'
f = 3; i = 13; j = 15	f = i ^ j	f = 2; i = 13; j = 15
f = 63; i = 11; j = 53	f = i ^ j	f = 62; i = 11; j = 53
a = 13; t = 13	t = t * a	a = 13; t = 169
b = 'ABACABA['; j = 3; l = 'AABABA'	l += b[j]	b = 'ABACABA['; j = 3; l = 'AABABAC'
f = '00001000'	r = f	f = '00001000'; r = '00001000'
c = 30; w = [4, 3, 2, 1, 1]; x = 2; y = 3	c = c + int(w[y]) * (x + 1)	c = 33; w = [4, 3, 2, 1, 1]; x = 2; y = 3
b = 24; w = 3	w = int(b ** 0.5)	b = 24; w = 4
p = 'cd'; u = {'c': 1}	u[p] = 1	p = 'cd'; u = {'c': 1, 'cd': 1}

Figure 4.3: Representative entry from the `Python State Changes` dataset. Each sample includes the pre-edit state (`start`), the applied edit fragment (`code`), and the resulting post-edit state (`end`).

The dataset offers realistic supervision for modeling code evolution, but it also contains noise: many examples include formatting changes, snippet extractions, or partial edits without full syntactic closure. This heterogeneity contributes to the difficulty encountered by the diff-aware encoder during downstream generation.

CodeXGLUE Code-to-Text: Supervised Generative Fine-Tuning

The Python subset of the *CodeXGLUE code-to-text* benchmark provides high-quality pairs of Python functions and human-written summaries. It is used both for supervised fine-tuning and for all evaluation splits, ensuring comparability with previous work.

Figure 4.4 shows a typical sample.

code string · lengths 75 2.04k 92%	code_tokens list · lengths 20 89 62%	docstring string · lengths 3 1.73k 98.7%
def sum_to_n(n, size, limit=None): #from http://stackoverflow.com/questions/2065553/python…	["def", "sum_to_n", "(", "n", ",", "size", ",", "limit", "=", "None", ")", ":", "#from…	Produce all lists of `size` positive integers in…
def nfa(self, nextstate): """Returns an initial state for an NFA""" if self.interval: mininterval…	["def", "nfa", "(", "self", ",", "nextstate", ")", ":", "if", "self", ".",…	Returns an initial state for an NFA
def nfa(self): """convert the expression into an NFA""" finalstate = State(final=True) nextstate =…	["def", "nfa", "(", "self", ")", ":", "finalstate", "=", "State", "(", "final",…	convert the expression into an NFA
def stddev(values, meanval=None): #from AI: A Modern Appproach """The standard deviation of a…	["def", "stddev", "(", "values", ",", "meanval", "=", "None", ")", ":", "#from AI…	The standard deviation of a set of values. Pass in the…
def save(self, filename, addnormalised=False): """Save a frequency list to file, can be loaded…	["def", "save", "(", "self", ",", "filename", ",", "addnormalised", "=",…	Save a frequency list to file, can be loaded later…
def output(self,delimiter = '\t', addnormalised=False): """Print a representation o…	["def", "output", "(", "self", ",", "delimiter", "=", "'\\t'", ",",…	Print a representation of the frequency list
def entropy(self, base = 2): """Compute the entropy of the distribution""" entropy = 0 if not…	["def", "entropy", "(", "self", ",", "base", "=", "2", ")", ":", "entropy", "=", "0",…	Compute the entropy of the distribution
def output(self,delimiter = '\t', freqlist = None): """Generator yielding formatted strings…	["def", "output", "(", "self", ",", "delimiter", "=", "'\\t'", ",", "freqlist",…	Generator yielding formatted strings expressing the tim…
def process(self, sourcewords, debug=False): """Process a list of words, passing it to the…	["def", "process", "(", "self", ",", "sourcewords", ",", "debug", "=", "False",…	Process a list of words, passing it to the server a…

Figure 4.4: Example entry from the `CodeXGLUE` Python code-to-text dataset, consisting of a Python function and its associated docstring.

Compared to the two contrastive datasets, CodeXGLUE is:

- **larger**,
- **cleaner**,
- **more stylistically consistent**,
- and **explicitly designed for summarization**.

For these reasons, it serves as the authoritative benchmark for evaluating the models developed in this thesis.

Comparative Overview

A concise comparison highlights the complementary roles of the three datasets:

- **CodeSearchNet** — clean, static, function-level supervision suitable for learning semantic equivalence.
- **Python State Changes** — noisy, dynamic, commit-level supervision capturing actual code evolution.
- **CodeXGLUE** — high-quality, summarization-oriented data for supervised fine-tuning and evaluation.

Together, these datasets support a multi-stage learning curriculum aligned with the methodological structure of the thesis.

Training Environment and Resources

All experiments were implemented in `PyTorch` using the Hugging Face `Transformers` and `Datasets` libraries. Training was executed on a single 8 GB NVIDIA GPU. Given this limitation, a sequence of memory-aware strategies—gradient checkpointing, mixed-precision computation (FP16), and gradient accumulation—was adopted to emulate larger effective batch sizes. Each fine-tuning run employed roughly 50 000 training instances and 2 000 validation examples drawn from the Python subset of the *CodeXGLUE code-to-text* dataset. For final evaluation, all metrics were computed on the 13 901-sample official test split to ensure comparability with prior literature.

The learning procedure was structured into successive stages: an initial fine-tuning phase of up to three epochs, followed by targeted continuation and polishing phases with lower learning rates. All runs used beam search decoding with four beams, a length penalty of 1.15, repetition penalty 1.05, and early stopping, yielding balanced summaries that avoided verbosity or redundancy.

Model Variants

The following encoder–decoder configurations were tested:

- **ED-Base** – Both encoder and decoder initialized from `microsoft/codebert-base`, representing the baseline.

- **ED-Code-Code** – Encoder initialized from the contrastively trained function-level encoder; decoder identical to baseline.
- **ED-Code-Diff** – Encoder initialized from the sequentially pretrained diff-aware model, which itself derives from the code–code encoder and is specialized for semantic changes between code versions.

All three models share identical decoding architectures, allowing direct attribution of performance differences to encoder representations rather than to language-model capacity.

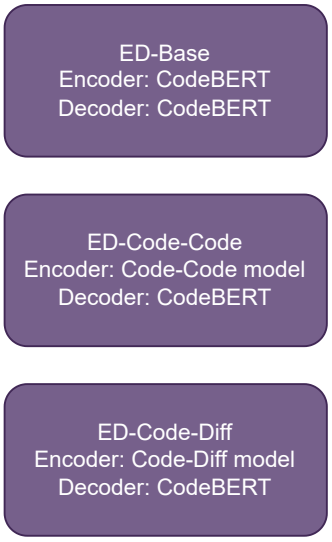


Figure 4.5: Block comparison of the three encoder–decoder configurations evaluated. All models share the same decoder; differences arise exclusively from encoder initialization strategies.

Evaluation Metrics

Model outputs were assessed using standard automatic metrics commonly employed in code summarization research:

- **ROUGE-L** – Measures the longest common subsequence between generated and reference texts, emphasizing structural overlap.
- **SacreBLEU** – Computes n-gram precision with brevity penalty; standardized via the `sacreBLEU` library to ensure reproducibility.
- **METEOR** – Combines precision, recall, and synonym matching at the word level, offering sensitivity to paraphrases.

- **BERTScore** – Estimates semantic similarity using contextual embeddings from a pretrained transformer, capturing meaning beyond surface tokens.
- **Average Length** – Reports the mean number of generated tokens, monitoring conciseness and verbosity.

Although automatic metrics remain imperfect proxies for semantic fidelity, they offer complementary insights into syntactic accuracy, content recall, and expressive fluency.

4.3 Quantitative Results

Table 4.1: Automatic evaluation metrics on the test split (n = 13 901).

Model	ROUGE-L	SacreBLEU	METEOR	BERTScore F1	Avg. Len.
ED (Base Encoder)	**0.2136**	**3.1401**	**0.1872**	**0.8440**	26.5
ED (Code–Code Encoder)	0.1928	2.7762	0.1765	0.8402	25.6
ED (Diff–Code Encoder)	0.1568	1.8514	0.1380	0.8254	21.0

The results in Table 4.1 reveal a nuanced picture. Contrary to the initial expectation, the baseline model marginally surpasses the contrastively pretrained variants across all metrics. While the **ED-Code-Code** model exhibits competitive performance and produces slightly shorter but coherent outputs, the **ED-Code-Diff** model clearly trails behind, reflecting the difficulty of transferring change-aware embeddings to the generative task.

These differences must be interpreted in light of dataset scale and pretraining coverage. The base encoder benefits from massive multilingual corpora, whereas the contrastive encoders—although semantically structured—were exposed to narrower and noisier data during pretraining. As a result, their representational geometry emphasizes structural and behavioral alignment rather than the broader linguistic fluency that automatic metrics tend to reward.

4.4 Qualitative Analysis

Quantitative measures, though informative, conceal subtleties of model behavior. A qualitative inspection of generated summaries further highlights the differences in fluency, semantic grounding, and lexical coherence across the three model variants.

Table 4.2: Representative qualitative samples from the test split.

Example 1
Code snippet:
`def cleanup_database_hook(self):`
Reference: Clean up database hook after it was used.
ED-Base: Clean the database connection after use.
ED-Code-Code: Clean up database after used.
ED-Code-Diff: Return a list of the database.
Example 2
Code snippet:
`def next_retry_datetime(self):`
Reference: Get datetime of the next retry if the task instance fails.
ED-Base: Return the datetime of the next retry if task fails.
ED-Code-Code: Get datetime of retry if task instance fails.
ED-Code-Diff: Set the task of the datetime.

The **ED-Base** model yields grammatically consistent yet sometimes generic descriptions, demonstrating strong linguistic priors but limited structural grounding. The **ED-Code-Code** model often matches or exceeds the baseline in semantic accuracy, indicating effective transfer of functional information from contrastive learning. By contrast, the **ED-Code-Diff** summaries are frequently shorter and semantically incomplete, reflecting representational drift introduced by diff-level pretraining.

4.5 Extended Discussion

Contrastive Learning in Generative Contexts

Contrastive encoders excel at discriminative and retrieval tasks, where relative distances define semantic similarity. In generative settings, however, the mapping from embeddings to sentences is mediated by the decoder, which must adapt to the structure of the embedding space. If fine-tuning capacity is limited, the decoder may fail to exploit the encoder's geometry fully, reducing the apparent benefits of contrastive pretraining.

Limitations of Diff-Aware Representations

Commit-level diffs often reflect a mixture of semantic and non-semantic edits. Refactorings, formatting changes, and partial code snippets introduce noise that weakens the supervision signal. Such instability compromises the ability of the diff-aware encoder to serve as a reliable basis for text generation, even though it may still be useful for tasks such as change classification or commit message generation.

Metric Sensitivity and Semantic Fidelity

Surface-level metrics like BLEU or ROUGE emphasize token overlap rather than conceptual correctness. Thus, even semantically aligned summaries may score lower if they paraphrase or reorder content. BERTScore mitigates this to an extent but remains influenced by decoder fluency. Future evaluations could incorporate human assessment or task-based comprehension tests.

4.6 Limitations and Future Directions

The evaluation must be interpreted considering several constraints: limited GPU memory, noisy contrastive datasets, and reliance on automatic metrics. Furthermore, code summarization benefits from high-quality paired data, which remains scarce. Future work may explore larger pretraining corpora, multi-language encoders, or reinforcement learning with human feedback. Hybrid approaches combining diff-aware learning with syntactic information from ASTs or data-flow graphs may also enhance semantic grounding.

4.7 Summary

The experimental results illustrate the complexity of bridging structured code representations with natural-language summarization. The baseline model achieves the highest automatic scores, reflecting its extensive pretraining. The contrastively trained code–code encoder remains competitive and improves semantic grounding, though its advantages do not fully translate under the current fine-tuning constraints. The diff-aware encoder underperforms in generation but offers insights into modeling code evolution, suggesting alternative downstream applications.

Overall, the experiments validate the methodological design while highlighting limitations of contrastive–generative integration under realistic training conditions. These findings inform the broader discussion presented in the final chapters, pointing toward future refinements in representation learning for code intelligence.

Chapter 5

Discussion and Future Works

5.1 Summary of Contributions

This research explored the intersection between contrastive representation learning and neural text generation for source-code understanding. Its central goal was to determine whether pretraining encoders through semantically structured objectives could improve downstream comment generation when integrated into encoder–decoder architectures. To this end, two forms of contrastive pretraining were investigated: a *function-level* approach capturing semantic equivalence between code fragments, and a *commit-level* approach modeling the evolution of code through version-control diffs.

The proposed framework introduced several methodological innovations:

- A sequential pretraining pipeline in which a code–code encoder serves as the foundation for a diff-aware encoder, promoting transfer of syntactic and structural knowledge.
- The integration of these pretrained encoders into encoder–decoder models for docstring generation, establishing a link between discriminative and generative learning paradigms.
- A progressive fine-tuning regime involving staged unfreezing and multi-phase continuation, designed to balance convergence speed with representational stability under hardware constraints.
- A systematic evaluation across quantitative metrics (ROUGE-L, SacreBLEU, METEOR, BERTScore) and qualitative inspection, providing an empirical basis for assessing contrastive influence on generative performance.

Together, these elements constitute a coherent methodological contribution that bridges the traditionally separate domains of contrastive representation learning and code summarization. Beyond its immediate outcomes, the project demonstrates how controlled pretraining pipelines can be implemented and analyzed under realistic computational limits, offering a reproducible experimental template for future research.

5.2 Interpretation of Results

The empirical evidence reveals a nuanced balance between theoretical promise and practical outcome. The baseline encoder–decoder initialized entirely from `microsoft/codebert-base` attained the highest overall scores across all automatic metrics, with ROUGE-L = 0.2136, SacreBLEU = 3.1401, METEOR = 0.1872, and BERTScore F1 = 0.8440. The code–code variant followed closely with slightly lower but comparable values, while the diff-aware model lagged behind substantially.

These findings suggest that the benefits of contrastive alignment, though conceptually sound, did not fully translate into measurable improvements in generative quality under the experimental conditions. Several interrelated factors help explain this outcome.

First, the **data scale and domain coverage** of CodeBERT's original pretraining far exceed those of the contrastive datasets used in this work. The contrastive stages, while semantically focused, exposed the encoder to fewer and more homogeneous examples, narrowing its lexical and syntactic repertoire. During fine-tuning, this limitation manifested as slightly reduced linguistic fluency, which metrics like BLEU and ROUGE heavily penalize.

Second, the **nature of the supervision signal** differs between the two paradigms. Contrastive learning enforces relative geometry in the embedding space, rewarding correct proximity relations rather than explicit reconstruction of text. Generative decoding, by contrast, demands token-level precision. The transformation from structured similarity to free-form generation may therefore require intermediate adaptation that was not explicitly modeled here.

Third, the **diff-aware stage** introduced additional complexity. Commit-level pairs from open repositories are inherently noisy: they contain formatting changes, partial edits, or auxiliary updates with no clear semantic correspondence. As a result, the encoder learned relational patterns that do not consistently map to meaningful linguistic cues, producing unstable generation behavior. Nonetheless, this limitation underscores a broader insight — that modeling *change* in code may serve purposes beyond comment generation, such as defect prediction or commit-message synthesis.

5.3 Methodological Reflections

From a methodological standpoint, the project illustrates both the potential and the fragility of transfer learning within software-language modeling. The sequential structure — function-level pretraining followed by diff-level specialization — proved effective for knowledge reuse, but also propagated upstream imperfections. This dependency highlights the importance of **robust foundation models** and of maintaining a clear conceptual separation between transferable syntax-level features and task-specific semantics.

The fine-tuning regimen, built around progressive unfreezing and learning-rate decay, proved valuable in stabilizing optimization on constrained hardware. Stage

A (decoder-only) allowed the language model to adapt without disrupting previously learned embeddings, while Stage B (partial unfreezing) enabled subtle encoder–decoder co-adaptation. Although this approach did not yield superior quantitative metrics, it produced smoother convergence curves and reduced overfitting, validating the practicality of staged adaptation for medium-scale training.

The evaluation also underscores the limitations of current **automatic metrics** in reflecting true semantic adequacy. Generated comments often captured the correct intent in paraphrastic form, yet received low BLEU or ROUGE scores due to lexical divergence. Future evaluations should include human judgment or task-oriented assessments, such as developer comprehension tests, to more faithfully measure usefulness.

5.4 Lessons Learned

Several broader lessons emerge from this investigation:

- **Data quality outweighs objective sophistication.** Clean, well-aligned examples of code and natural language remain the decisive factor in downstream performance. Even advanced learning objectives cannot compensate for noisy supervision.
- **Representation transfer is not linear.** Features optimized for similarity may not directly benefit generative tasks; additional alignment layers or adapters could facilitate smoother transition between embedding and decoding spaces.
- **Evaluation must match purpose.** For a human-facing task like code summarization, fluency and clarity matter as much as lexical overlap. A hybrid evaluation combining automatic and qualitative measures provides the most informative picture.
- **Incremental experimentation is essential.** Conducting the study in progressive stages — from base models to contrastive variants, from full to mini fine-tuning — enabled consistent debugging, reproducibility, and interpretability of results.

These insights inform not only the interpretation of this work but also broader research in code intelligence. They demonstrate the importance of viewing representation learning and generation as interdependent components of a larger semantic modeling ecosystem.

5.5 Future Directions

The work opens multiple avenues for further investigation. First, the most immediate extension involves **scaling contrastive pretraining** to larger and more

diverse repositories. Datasets covering multiple programming languages, richer documentation, and aligned commit metadata would provide stronger semantic signals. Self-supervised mining of functionally equivalent snippets through unit-test behavior or symbolic execution could reduce manual curation costs.

A second direction concerns the **integration of structural information**. Augmenting the encoder with abstract syntax trees (ASTs), control-flow graphs, or data-dependency representations could improve sensitivity to program semantics beyond surface tokens. Graph-based contrastive objectives, in which subtrees or code paths act as positive pairs, might better capture hierarchical relationships.

Third, future research should explore **multitask and multi-objective training**. Rather than treating diff modeling and summarization as sequential phases, they could be optimized jointly. A shared encoder could learn to serve both change detection and description generation, encouraging richer cross-task representations. Techniques such as adapter fusion or soft parameter sharing can enable such hybridization without excessive memory cost.

Fourth, **human-in-the-loop learning** represents a promising paradigm. Developers could rate or edit generated comments during routine development, providing implicit feedback for reinforcement or contrastive tuning. This iterative loop would align model behavior more closely with practical expectations and domain-specific conventions.

Finally, **evaluation methodologies** require rethinking. Automatic metrics remain useful but should be complemented by semantic similarity measures derived from large language models, as well as by human assessments of correctness, coverage, and readability. Establishing shared benchmarks that reflect real-world developer usage would strengthen comparability across studies.

5.6 Concluding Remarks

Despite its mixed quantitative results, this work makes a tangible contribution to the understanding of how semantic pretraining interacts with code generation. It demonstrates that function-level contrastive learning can yield meaningful structural representations and that modeling code evolution, while challenging, introduces a valuable new perspective on temporal semantics in software. The study also exemplifies the process of designing, training, and evaluating complex neural architectures within realistic computational and temporal constraints — an experience increasingly common in applied research.

In a broader sense, the research illustrates the delicate interplay between language, logic, and learning. Code is both a linguistic and a formal object: it requires models to reason about syntax, semantics, and intent simultaneously. Bridging this gap between representation and expression remains one of the central challenges of machine learning for software engineering. The methods, experiments, and reflections presented here contribute a small but coherent step toward that long-term objective.

Chapter 6

Conclusions

6.1 Closing Overview

The work presented in this thesis set out to explore whether contrastive representation learning can enhance the capacity of neural models to generate meaningful natural-language summaries of source code. Grounded on the `CodeBERT` architecture, the study designed and implemented a two-stage pretraining framework in which encoders were first trained to learn semantically structured embeddings of code and then integrated into encoder–decoder architectures for comment generation. Two complementary forms of contrastive supervision were examined: the *function-level* approach (**code–code**) and the *commit-level* or **code–diff** approach. Together, they embody two distinct views of software semantics — one static, centered on equivalence, and one dynamic, centered on change.

6.2 Summary of Findings

Empirical evaluation across standard benchmarks yields a nuanced picture. The baseline encoder–decoder derived directly from `microsoft/codebert-base` achieved the strongest overall scores in ROUGE-L, SacreBLEU, METEOR, and BERTScore, benefiting from its broad linguistic prior and large-scale pretraining. The function-level contrastive model (**ED-Code-Code**) performed competitively, producing coherent and faithful docstrings that frequently captured intent even when lexical overlap was low. Its results confirm that semantic alignment between functionally similar snippets enhances representational robustness and supports better content selection during generation. In contrast, the commit-level contrastive model (**ED-Code-Diff**) underperformed, generating shorter and less consistent sentences. This outcome reflects the intrinsic noise and partial context of commit-level data, where differences may not correspond to clear semantic transformations.

Taken together, the findings indicate that contrastive pretraining offers clear structural benefits at the representation level but that these benefits do not automatically translate into improvements in token-level generation metrics. The alignment between embedding geometry and linguistic fluency remains a challenging bridge to

cross, particularly under limited data and compute regimes.

6.3 Scientific and Practical Implications

Beyond numeric scores, the project contributes to a deeper understanding of how neural representations of code can be shaped through intermediate objectives. It shows that contrastive learning can successfully organize code embeddings according to functional semantics and that such organization can be transferred, at least partially, to downstream tasks. From a methodological perspective, the research demonstrates the viability of sequential pretraining pipelines — where knowledge is first consolidated at the function level and then specialized for higher-order relations — even when executed on modest hardware through progressive unfreezing and gradient checkpointing.

From a practical standpoint, the work highlights the challenges of applying machine-learning models to real software artifacts. Code repositories are heterogeneous, comments are inconsistent, and commit histories often encode incidental rather than semantic changes. Effective automation of documentation therefore requires not only better architectures but also cleaner and more context-aware data collection strategies. In this sense, the results invite a reconsideration of data engineering as a core component of model design for software intelligence.

6.4 Limitations

Several limitations frame the interpretation of these results. The experiments were conducted on a single-language corpus (Python) and under tight computational constraints, which limited hyper-parameter exploration and full convergence. Evaluation relied primarily on automatic metrics that emphasize lexical overlap rather than semantic adequacy. Moreover, the diff-level pretraining data, drawn from raw commit histories, contained substantial noise that likely hindered the model's ability to learn consistent relational semantics. These constraints do not undermine the validity of the findings but delineate the boundaries within which they should be generalized.

6.5 Future Perspectives

The research opens multiple avenues for continuation. Future studies could scale contrastive pretraining to larger and cleaner corpora, possibly integrating signals from execution traces or unit-test outcomes to anchor semantics more firmly in program behavior. Combining structural encodings — such as abstract syntax trees or data-flow graphs — with textual embeddings could yield hybrid representations that better capture code logic. Joint multitask frameworks in which contrastive and generative objectives are optimized simultaneously may bridge the current gap between representation learning and language generation. Finally, incorporating human feedback — for example through developer ratings or in-IDE interactive

correction — could guide models toward more pragmatic, readable, and contextually relevant documentation.

6.6 Final Remarks

In conclusion, this thesis contributes both methodologically and empirically to the growing field of neural software understanding. It demonstrates that the principles of contrastive learning, long established in vision and natural-language processing, can be meaningfully applied to source code representation. It also reveals the limits of this transfer when confronted with the complexity and noisiness of real software evolution. While the quantitative advantages of contrastive pretraining for comment generation remain modest under current constraints, the conceptual integration of representational and generative learning represents a significant step forward. The findings underscore a broader message: meaningful progress in code intelligence will emerge not from ever-larger models alone but from the principled design of objectives, datasets, and evaluation strategies that reflect the multifaceted nature of programming itself.

References

[1] Annibale Panichella. "The impact of code comments on software maintenance: A systematic mapping study". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 507–511 (cit. on p. 1).

[2] Xing Hu et al. "Deep code comment generation". In: *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE. 2018, pp. 200–210 (cit. on pp. 1, 10).

[3] Zhen Zhang et al. "A Review on Automatic Code Comment Generation: Datasets, Methods, and Challenges". In: *Empirical Software Engineering* 29.3 (2024), p. 58. DOI: 10.1007/s10664-024-10553-6 (cit. on pp. 2, 10).

[4] Ashish Vaswani et al. "Attention is All You Need". In: *Advances in Neural Information Processing Systems* 30 (2017) (cit. on pp. 3, 8).

[5] Zhangyin Feng et al. "CodeBERT: A pre-trained model for programming and natural languages". In: *arXiv preprint arXiv:2002.08155* (2020) (cit. on pp. 3, 8).

[6] Yue Wang, Weishi Wang, Shafiq Joty, et al. "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation". In: *EMNLP*. 2021, pp. 8696–8708 (cit. on pp. 3, 8, 10).

[7] Wasi Ahmad, Saikat Chakraborty, and Baishakhi Ray. "PLBART: Pre-training language models for text-to-code generation". In: *ACL*. 2021, pp. 726–740 (cit. on pp. 3, 9, 10).

[8] Marie-Anne Lachaux et al. "Unsupervised translation of programming languages". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 20601–20611 (cit. on p. 3).

[9] Daya Guo et al. "UniXCoder: Unified cross-modal pre-training for code representation". In: *ACL* (2022) (cit. on pp. 3, 9).

[10] Wasi Ahmad et al. "A transformer-based approach for source code summarization". In: *ACL Findings* (2020) (cit. on p. 4).

[11] Tianyu Gao, Xingcheng Yao, and Danqi Chen. "SimCSE: Simple contrastive learning of sentence embeddings". In: *EMNLP*. 2021, pp. 6894–6910 (cit. on pp. 4, 10).

[12] Jason Chiu et al. “Contrastive learning for code representation”. In: *ICSE*. 2023 (cit. on pp. 4, 10).

[13] Daya Guo et al. “GraphCodeBERT: Pre-training code representations with data-flow”. In: *ICLR*. 2021 (cit. on pp. 4, 8).

[14] Veselin Raychev, Martin Vechev, and Eran Yahav. “Code completion with statistical language models”. In: *PLDI*. 2014 (cit. on p. 7).

[15] Miltiadis Allamanis et al. “A survey of machine learning for big code and naturalness”. In: *ACM Computing Surveys* (2018) (cit. on p. 7).

[16] Srinivasan Iyer et al. “Summarizing source code using neural attention”. In: *ACL*. 2016 (cit. on pp. 7, 10).

[17] Miltiadis Allamanis et al. “Learning to represent programs with graphs”. In: *ICLR* (2018) (cit. on p. 7).

[18] Jie Zhang et al. “A graph-based neural model for code summarization”. In: *ICSE*. 2019 (cit. on p. 7).

[19] Hamel Husain et al. “CodeSearchNet Challenge: Evaluating the state of semantic code search”. In: *NeurIPS Workshop*. 2019 (cit. on p. 8).

[20] Yue Wang et al. “CodeT5+: Open code large language models for code understanding and generation”. In: *ACL* (2023) (cit. on pp. 9, 10).

[21] Ming Ding et al. “Contrastive learning for source code representation”. In: *IEEE Transactions on Software Engineering* (2022) (cit. on p. 10).

[22] Xuejun Hu et al. “Retrieval-augmented code summarization via cross-modal learning”. In: *ICSE*. 2023 (cit. on pp. 10, 11).

[23] Emon Haque, Md Rafiqul Hossain, and Baishakhi Ray. “Improving code summarization through semantic consistency”. In: *Empirical Software Engineering* (2023) (cit. on p. 11).

[24] Alexander LeClair and Collin McMillan. “Improved code summarization via a graph neural network”. In: *ICPC*. 2020 (cit. on p. 11).